Automated Parallelization to Improve Usability and Efficiency of Distributed Neural Network Training

Nathaniel J. Grabaskas

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2018

Committee:

Munehiro Fukuda, Chair

Erika Parsons

Clark Olson

Program Authorized to Offer Degree:

Computing & Software Systems

© Copyright 2018

Nathaniel J. Grabaskas

University of Washington

Abstract

Automated Parallelization to Improve Usability and Efficiency of Distributed Neural Network Training

Nathaniel J. Grabaskas

Chair of the Supervisory Committee:
Munehiro Fukuda
Computing & Software Systems

The recent success of Deep Neural Networks (DNNs) [1] has triggered a race to build larger and larger DNNs [2]; however, a known limitation is the training speed [3]. To solve this speed problem, distributed neural network training has become an increasingly large area of research [4], [5]. Usability, the complexity for a machine learning or data scientist to implement distributed neural network training, is an aspect rarely considered, yet critical. There is strong evidence growing complexity has a direct impact on development effort, maintainability, and fault proneness of software [6]–[8]. We investigated, if automation can greatly reduce the implementation complexity of distributing neural network training across multiple devices without loss of computational efficiency when compared to manual parallelization. Experiments were conducted using Convolutional Neural Networks (CNN) and Multi-Layer Perceptron

(MLP) networks to perform image classification on CIFAR-10 and MNIST datasets. Hardware consisted of an embedded, four node NVIDIA Jetson TX1 cluster. Torch Automatic Distributed Neural Network (TorchAD-NN) reduces the implementation complexity of data parallel neural network training by more than 90% and providing components, with near zero implementation complexity, to easily parallelize all or only select fully-connected neural layers.

TABLE OF CONTENTS

List of Figures iv

List of Tables vi

Chapter 1. Introduction 1

1.1 Current Approach and Limitations 2

1.1.1 Data and Model Parallelism 2

1.1.2 Data and Training Goals 2

1.1.3 Limitations 3

1.2 Motivation 4

1.3 Research Goal and Criteria 6

1.4 Approach Overview and Contributions 7

1.5 Outline 8

Chapter 2. Background And Related Work 9

2.1 Background 9

2.1.1 ANN, DNN, and CNN 9

2.1.2 Embedded Systems 13

2.2 Conventional Approaches And Comparison 13

2.2.1 Data Parallelism Comparison 13

2.2.2 Model Parallelism Comparison 15

Chapter 3. Methods 18

3.1 Platforms for Research ... 18

3.1.1 Torch vs TensorFlow ... 18

3.1.2 Torch and TorchMPI ... 21

3.1.3 Hardware Setup – Cobra Cluster ... 22

3.1.4 Datasets ... 23

3.1.5 Networks ... 24

3.1.6 Training Method ... 25

3.2 System Design ... 25

3.2.1 Data Parallel Module ... 26

3.2.2 Model Parallel Module ... 30

3.2.3 Development Lifecycle ... 35

3.3 Metrics ... 36

3.3.1 Usability (Halstead's Complexity Measures) ... 36

3.3.2 Metrics for Performance Evaluation ... 38

3.4 Challenges ... 39

3.4.1 Synchronization ... 39

3.4.2 Spatial Convolution Layer Parallelism ... 40

Chapter 4. Evaluation ... 42

4.1 Experiment Setup ... 42

4.1.1 Data Parallel Setup ... 42

4.1.2 Model Parallel Setup ... 43

4.1.3 Complexity Evaluation Example ... 44

4.2 Main Takeaways ... 46

4.2.1 Data Parallel Complexity (Usability) ... 47
4.2.2 Data Parallel Performance ... 49
4.2.3 Data Parallel Unique Attribute (Batch Size) ... 52
4.2.4 Data Parallel Unique Attribute (Pre-processing) ... 55
4.2.5 Model Parallel Complexity (Usability) ... 57
4.2.6 Model Parallel Performance ... 60
4.2.7 Model Parallel Unique Attribute (Model Size) ... 65

Chapter 5. Conclusion ... 67

5.1 Result Statement ... 67
5.2 Stakeholder Effect ... 68
5.3 Personal Effect ... 69
5.4 Limitations ... 70
5.5 Future Work and Next Steps ... 71

Bibliography ... 72

Appendix A – Source Code ... 75

Appendix B – Neural Network Training Setup ... 90

Appendix C – Test Results (Data Parallel) ... 92

Appendix D – Test Results (Model Parallel) ... 94

LIST OF FIGURES

Figure 2.1. Shows a DNN of five layers. L1 accepts the individual pixel values, L2 identifies edges of objects within the image, L3 identifies connected edges, L4 uses these combined edges to identify features from the image, and then L5 uses these features to determine what the image contains [27]. 11

Figure 2.2. Architecture of LeNet-5 [21]. *Demonstrates how multiple convolution and subsampling layers work together to increase the number of feature maps and decrease the resolution of the image. Which is then fed into a fully-connected multilayer network to form the complete CNN.* 12

Figure 3.1. High level system architecture view 26

Figure 3.2. In data parallelism the data is partitioned equally across all nodes and each node gets a full copy of the network model. During training the gradients are synchronized at the end of each training batch. At the end of training each node contains a complete copy of the trained model. 27

Figure 3.3. Flow chart outlining the four stages of automated data parallelization 30

Figure 3.4. This model demonstrates input of size 1024 split between two nodes, each node takes half (512 of the inputs). Each node generates its own output of size 10 and is then synchronized between the nodes using allReduce. 31

Figure 3.5. This diagram demonstrates the splitting and synchronization during forward and backward propagation using MLP2. Input starts as 1024, is narrowed to 512 between 2 nodes, and each generates their respective 2048 output. Each node then uses allReduce() to synchronize their outputs and proceeds through the Tanh component. After Tanh, the input is again narrowed from 2048 to 1024 on each node. AllReduce() is used again to synchronize each nodes output of size 10. This concludes the forward propagation. Backward propagation is the reverse, the output of 10 is then updated back to 1024, this 1024 is synchronized between the nodes with allReduce, which is fed into the Tanh layer. Here the Tanh layer must expand the 1024 to 2048 since the next layer is expecting this size output. The 2048 is then updated back up into each nodes 512 and again synchronized

using allReduce(). Each full cycle of forward and backward propagation requires 4 allReduce() calls. 32

Figure 4.1. Data from lines D.9, D.10, and D.11 of Table 4.16, plotted to show the comparison between accuracy and efficiency when batch size is varied between 100, 500, and 1000. 54

Figure 4.2. Training time using Auto SGD Train() with MLP3 and 4 nodes. Demonstrates training time increases as nodes are added due to communication overhead. 64

LIST OF TABLES

Table 1.1. Code snippets representing the dataset and model for CPU training and the code necessary to convert the training to GPU. 5

Table 2.2. Human vs Artificial Neuron Similarity Chart [24] 10

Table 3.3. Lua vs. Python by category [38]–[40] 19

Table 3.4. Wave2D Simulation results using (4) TX1s and (4) UWB-320 machines. Time is the average from 10 test runs using a 576 x 576 matrix for 500 iterations. 22

Table 3.5. MPInitialLinear.updateOuput() source code 33

Table 3.6. HCM uses the following parameters: 37

Table 3.7. HCM provides the following metrics [52]*:* 38

Table 3.8. Code snippet from SpatialConvolutionMM_updateOutput() [53] *demonstrating outputHeight and outputWidth are dependent on multiple parameters and do not scale proportionally with inputHeight and inputWidth.* 41

Table 4.9. CIFAR Backward() Manual Parallelization Comparison 44

Table 4.10. CIFAR Backward() Auto Parallelization Comparison 45

Table 4.11. CIFAR SGD Train() Manual Parallelization Comparison 45

Table 4.12. CIFAR SGD Train() Auto Parallelization Comparison 46

*Table 4.13. Effort and Time measurements for data parallel implementations using HCM.*48

Table 4.14. MNIST CPU Data Parallel Parallelization Performance. 50

Table 4.15. CIFAR GPU Data Parallel Parallelization Performance. Five training iterations. 51

Table 4.16. CIFAR GPU Data Parallel Parallelization Batch Size Comparison. One training iteration. 53

Table 4.17. CIFAR GPU Data Parallel Parallelization comparison between parallelization and reduced dataset on a single node. 54

Table 4.18. MNIST pre-processing time comparison across 1 to 4 nodes. 56

Table 4.19. CIFAR pre-processing time comparison across 1 to 4 nodes. 56

Table 4.20. Lines of Code necessary for each variation of model parallelism. 59

Table 4.21. MLP3 CPU model complexity SGD Train() compared with speedup efficiency trained on MNIST dataset. 61

Table 4.22. MLP and CNN GPU SGD Train() speedup efficiency trained on MNIST dataset. 63

Table 4.23. MLP3 GPU model complexity SGD Train() compared with speedup efficiency trained on MNIST dataset. 66

ACKNOWLEDGEMENTS

First, I would like to thank my committee chair, Dr. Munehiro Fukuda, for his attention to detail, thoughtful insight, and immense knowledge. His quiet nodes of approval and long pauses for a thoughtful response will be missed. He helped me take a step back from my work and carefully plan the next steps.

Besides my chair, I would also like to thank the rest of my thesis committee: Dr. Erika Parsons and Dr. Clark Olson, for their encouragement, insight, and difficult questions.

Last but not least, I want to thank my wife: Carol, without her in my corner pushing me into the fight I never would have overcome the challenges of the past two years.

`

DEDICATION

To my ever-supportive wife, Carol, and my little engineer, Zora.

Chapter 1. INTRODUCTION

Recently, Deep Neural Networks (DNNs) have seen tremendous success [1] in a wide range of applications. This success has triggered a race to build larger and larger DNNs [2], these in turn need more and more data for training, to solve problems in quickly growing fields of applications [9]. Increasing the scale of deep learning, with respect to the number of training examples, the number of model parameters, or both, can drastically improve ultimate classification accuracy [10]; however, a known scaling limitation is the training speed [3]. A high-accuracy DNN model can take weeks to train on a modern GPU [11]. Jeffrey Dean of Google in his recent keynote address stated [12], training time is a key challenge at the root of the development. To solve this speed problem, distributed neural network training has become an increasing large area of research [4], [5].

There are many research challenges with distributed neural network training, most commonly seen are increasing speed-up efficiency and maintaining network accuracy [13]–[15]. Usability, the complexity for a machine learning user or data scientist to implement distributed neural network training, is an aspect rarely considered, yet we consider equally as critical. The effect of complexity on software maintenance and fault-proneness has been studied extensively and there is sufficient evidence growing complexity has a direct impact on development effort, maintainability, and fault proneness of software [6]–[8]. We seek to diminish this implementation complexity by automating the process needed to take neural network training from a single computing node to multiple nodes. Torch Automatic Distributed Neural Network (TorchAD-NN) training, improves usability and maintains speedup of distributed neural network training.

1.1 Current Approach and Limitations

These subsections cover the types of neural network parallelism used, types of training tasks, and limitations.

1.1.1 *Data and Model Parallelism*

Neural network parallelism comes in two distinct flavors; data and model parallelism [3], [5], [9], [16]. Data parallelism is where each worker (i.e., TX1 node) gets an equal subset of the data, a complete copy of the neural network, and then workers communicate by exchanging weight gradient updates throughout the training. Model parallelism is the case where each worker gets the complete dataset, a partition of the neural network, and workers communicate by synchronizing output weight from each layer of neurons.

1.1.2 *Data and Training Goals*

Data comes in many shapes and formats, with many different purposes. In our solution, we seek to be independent from the type of data. The only assumption is the data is in some form of distinct instances.

The purposes of training are many, such as categorizing an entire image (Image classification), identifying objects within an image (Object identification), matching individual faces within an image or video (facial recognition), ability to identify spoken or written words or phrases (Natural Language Processing), click fraud detection, spam email detection, predictive sales analytics, content ranking, and more. With many purposes for data and neural networks, it was important to clearly identify which to use. We chose two different public datasets (CIFAR

and MNIST, see Chapter 3 for more details) both with the same training purpose of image classification.

Computers are unable to understand and process images in the way we do, they are only able to perform computations on numbers. There is a transformation that occurs between the original image and what the neural network accepts as input. These samples can be converted to greyscale where just one value represents each pixel, or RGB where each pixel has three values associated with it. With these images transformed to numbers, we teach the computer what a cat, a dog, a bird, etc. look like. This must be complete before the computer can recognize a new image. The more cats the computer sees, the better it gets at recognizing cats. This is known as supervised learning where the computer starts recognizing patterns present in cat pictures that are absent from other ones and starts building its own cognition [17]. From this learning, the computer can label a given image as a cat picture or an airplane pictures. This is known as image classification.

1.1.3 *Limitations*

With Torch [18] and TorchMPI [19] (publicly available libraries), parallelization of a given network and dataset can already be achieved. Torch provides the necessary neural network and CUDA frameworks to conduct training on the GPU, and TorchMPI executes inter-node communication from within Torch and adds synchronization functions.

For the rest of the paper, manual parallelism refers to manually adding the necessary code to take neural network training from a single node to multiple nodes. This uses the neural network and TorchMPI libraries as they stand without modification. Automated parallelization refers to using our TorchAD-NN library to take neural network training from a single node to multiple nodes.

The difficulty is this manual parallelization requires significant effort and knowledge from the user, discussed further in Section 4.2.1 and 4.2.5. This effort (or implementation complexity) is a major concern as the effect of complexity on software development has been studied extensively and there is sufficient evidence increased complexity has a direct impact on development effort, maintainability, and fault proneness of software [8]. Usability and complexity are used interchangeably in our research. They refer to the difficulty of implementing distributed neural network training from the viewpoint of the user.

Additionally, manual parallelization is not generally applicable to all networks and datasets, even once achieved it will need to be adjusted to fit each new model or dataset. Without sufficient testing after implementing manual parallelization, there is no guarantee efficient parallelization has been achieved.

1.2 MOTIVATION

Problem: *Our initial experience was manual parallelization took 25 hours and 190 Lines of Code (LoC) to distribute neural network training across 4 nodes.*

The inspiration for this research came from observing the ease in which neural network training was converted from CPU training to GPU training. This ease is demonstrated in Table 1.1, where a dataset and model can be converted to GPU training simply by adding the ":cuda()" function at the end. We sought to bring this high level of usability to distributed neural network training. Decreasing the complexity on neural network parallelization reduces development effort and mitigates fault proneness [20].

Table 1.1. Code snippets representing the dataset and model for CPU training and the code necessary to convert the training to GPU.

CPU Training	GPU Training
`Dataset`	`Dataset:cuda()`
`Model`	`Model:cuda()`

This work is aimed at helping those in small departments, scientists working in fields unrelated to computer science, and those working in machine learning as a hobbyist who lack the funding and access to large-scale GPU clusters. TorchAD-NN also targets machine learning and data scientists who do not have a strong foundation in software engineer and distributed computing. For those scientists, access to current distributed neural network training comes with a high level of implementation complexity. This research aims to give the powerful capabilities of distributed neural network training without requiring expertise and in-depth knowledge of distributed systems programming.

The automation is intended to work across all networks, training, and datasets. However, time and resources prevent extensive testing to completely confirm the solution is general. To mitigate this limitation, great care was taken to choose datasets and networks to accurately represent the general population. We chose two CNNs and two MLPs to test over two datasets, MNIST [21] and CIFAR [22].

Considering many classification systems need to be used in devices such as smart phones, robotics, or autonomous drones the limited resources environment is a major problem. Certain experiments were unable to be conducted due to the limited resources environment. Specifically, the amount of memory on the device (4GB) limits the complexity of a neural network or dataset to be used and LuaJIT further limits to 2GB. The maximum size of data and networks possible

are used. CIFAR-10 and MNIST provide a solid foundation to prove our module is highly capable.

1.3 RESEARCH GOAL AND CRITERIA

Hypothesis: *Automation can greatly reduce the implementation complexity of distributing neural network training across multiple devices without loss of computational efficiency when compared to manual parallelization.*

We identified three constraints our research solution must operate within. These ensure a solution has true value to the user not just for the sake of academic purpose.

- Automated parallelization solution must significantly increase usability (meaning decrease implementation complexity) while maintaining or improving efficiency.
- Solution must be applicable to all general neural networks and datasets. Significant modifications to datasets or networks should not be required to run the automated parallelization module.
- Framework must be lightweight enough to load and operate on standard and resource constrained environments (Jetson TX1 SoCs).

To ensure research success and limit time wasted, we established a strict scope and constantly reevaluated our project to prevent scope creep. This reduces resource waste and contributes to project success and timeliness.

- CPU and GPU methods are tested; however, the focus is on GPU training.
- We use existing neural networks and datasets.

- Our focus is on supplementing the TorchMPI framework regarding implementation complexity.
- The embedded platform is the Jetson TX1 Developer Board.

Complexity reduction goals were set based on our initial observations of manual parallelism and GPU implementation complexity. GPU implementation complexity was observed to be approximately 80% less than manual parallelization. We set our goal for automated parallelization complexity at this same 80% reduction to be as easy as GPU parallelization. In total, our final solution met the following criteria:

- Torch Automatic Distributed Neural Network (TorchAD-NN) training module released to the public.
- Reduce complexity of Data Parallel implementation by at least 80%.
- Reduce complexity of Model Parallel implementation by at least 80%.
- Maintain parallelization speedup efficiency.
- Add Stochastic Gradient Engine (SGD) engine support.

1.4 Approach Overview and Contributions

Experiment setup consists of a cluster of four Jetson TX1 hybrid CPU-GPU Systems on a Chip (SoC) for development and experiments. TX1s were chosen as the development hardware due to their low cost and well-established support system. The existing machine learning frameworks and communication protocols of Torch and TorchMPI were chosen. These two packages were chosen due to ease of debugging and a small footprint on the device when compared to their competitors, this is discussed in more detail under Chapter 3 under Existing Systems. As the

focus is on developing the framework for parallelization, publicly available networks and datasets were chosen for testing.

Torch Automatic Distributed Neural Network (TorchAD-NN) allows users who are not familiar with distributed systems programming to utilize distributed learning without the need for extensive programming knowledge. The main contribution is reducing the implementation complexity of data parallel neural network training by more than 90%. And providing components to easily parallelize all or only selected linear based layers of a neural network. Both are accomplished without reduction of efficiency when compared to a manual parallel solution.

1.5 OUTLINE

The Master's thesis is organized as follows. The next chapter covers related work to Neural Network parallelization, embedded systems, and comparable machine learning libraries. Chapter 3, covers existing systems, test environment setup, and module design. Test results and the meaning of results are covered Chapter 4. In conclusion, Chapter 5 states research value added and future work.

Chapter 2. BACKGROUND AND RELATED WORK

This chapter provides background information and related research. Background information gives a brief introduction to Artificial Neural Networks, Deep Neural Networks, and Convolutional Neural Networks. Section 2, discusses existing machine learning frameworks and research then compares their data parallel and model parallel approaches to ours.

2.1 BACKGROUND

This subsection introduces neural networks. Starts with the foundation of human neurons and how they inspired software neurons in Artificial Neural Networks. From there it is discussed what composes a Deep Neural Network and how Convolutional Neural Networks accomplish the more specific task of visual analysis. Finally, embedded system applications and their importance are covered.

2.1.1 *ANN, DNN, and CNN*

To understand Convolutional Neural Networks (CNN), we need to start at the foundation by explaining Artificial Neural Networks (ANN) and then define Deep Neural Networks (DNN). ANNs, as the name suggests, are inspired by the neurons in the human brain. This paradigm seeks to mimic the way the neurons in the human brain process information [23]. In the human brain, each biological neuron consists of a cell body, dendrites which bring information into the cell and an axon which transmits information out of the cell. A neuron fires along its axon when the collective effect of its inputs reaches a certain threshold. The axon from one neuron can influence the dendrites of another neuron across junctions called synapses. Some synapses

generate a positive effect, which encourages its neuron to fire, and others produce a negative effect, which discourages the neuron from firing.

An ANN is a network of interconnected software neurons. These software neurons consist of a processing element which has many input connections, each with an associated weight, a transfer function which determines the output, given the weighted sum of the inputs, and the output connection itself [24]. These similarities are shown in Table 2.2. The creation of a heavily interconnected network of these neurons form the ANN and solve problems through learning. The training of an ANN is perhaps the most important aspect. Information with known answers (ground truth) is fed through the ANN and the input weight and activation of each neuron is adjusted to move the network state towards being able to correctly identify all the training data answers [23], [25].

Table 2.2. Human vs Artificial Neuron Similarity Chart [24]

Human	**Artificial**
Neuron	Processing Element
Dendrites	Combining Function
Cell Body	Transfer Function
Axons	Element Output
Synapses	Weights

DNNs are composed of multiple layers of ANN neurons. Many experts [26] define them as having an input layer, an output layer, and at least one hidden layer in between. Each layer performs specific types of sorting and ordering in a process some refer to as “feature hierarchy” [26]. Demonstrated in Figure 2.1, the multiple layers of a DNN allow the network to identify

individual and combined elements of an image to understand what a "dog" may look like, this is outlined by Wolchover [27].

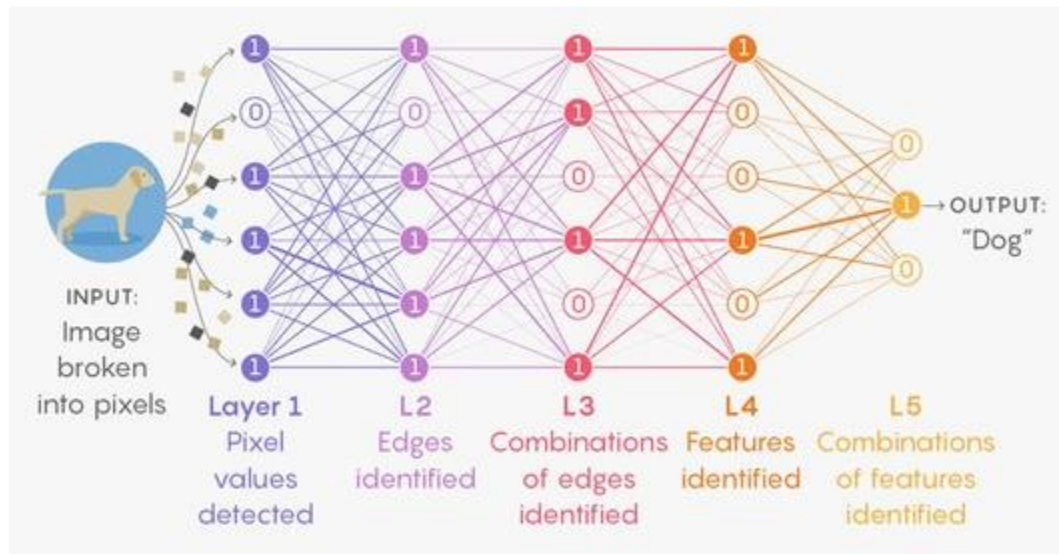

Figure 2.1. Shows a DNN of five layers. L1 accepts the individual pixel values, L2 identifies edges of objects within the image, L3 identifies connected edges, L4 uses these combined edges to identify features from the image, and then L5 uses these features to determine what the image contains [28].

CNNs are a specialized type of DNN which specialized in image processing. A CNN is comprised of one or more convolution layers and then followed by one or more fully connected layers as in a standard multilayer neural network [29]. In traditional pattern recognition you have a feature extractor and then a classifier. Rather than using a hand-design feature extractor, convolution layers serve as a feature extractor than can be trained using a dataset.

Convolution layers were inspired by the human visual cortex where cortical neurons respond to stimuli only in a restricted region of the visual field. CNNs mimic this behavior by forcing the extraction of local features by restricting inputs from the previous layer to a small local neighborhood. Because these connections are local they can extract elementary visual

features such as edges, endpoints, corners, etc. Subsequent layers combine these features to detect higher order features. A plane in which all the units share the same weights produce output which is known as a “feature map”. Because these units share the same weights they are constrained to perform the same operation on different parts of the image. Several feature maps compose a complete convolution layer.

Once a feature has been detected, its exact location is less important, its position relative to other features is what becomes significant. In fact, its exact position can be harmful to the network training. To reduce the position precision, a sub-sampling or pooling layer is used. Generally, these layers reduce the resolution of the feature maps and thereby reduce the sensitivity to shifts or distortions in the image. Multiple layers of alternating convolution and pooling increase the number of feature maps and reduce the spatial resolution, shown in Figure 2.2. At the end of the network, a fully-connected multilayer model uses these feature maps to classify the image [21].

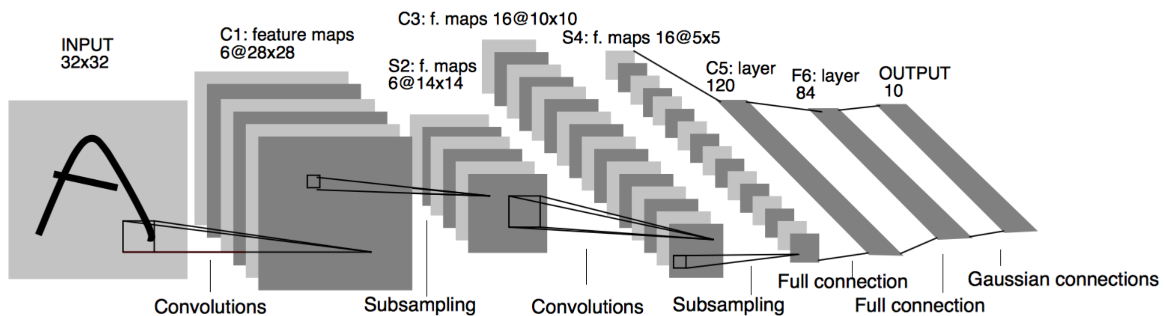

Figure 2.2. Architecture of LeNet-5 [21]. *Demonstrates how multiple convolution and subsampling layers work together to increase the number of feature maps and decrease the resolution of the image. Which is then fed into a fully-connected multilayer network to form the complete CNN.*

2.1.2 *Embedded Systems*

To prove the relevance of our automation modules, it is imperative we use publicly available datasets and networks directly related to current embedded applications. Object detection and image classification are essential to robotics [30] and autonomous vehicles [31]. Continued improvements to convolutional neural networks in recent years [2], [32] have made these possible on embedded systems.

It is well known, object detection requires more computation power and memory than image classification making it especially difficult to implement effectively on embedded systems [33]. Since our research is focused on increasing the usability of the framework, we chose to work only with image classification.

Image classification uses the Canadian Institute For Advanced Research (CIFAR) 10 [22] dataset with 10 classes and 5,000 instances per class. And also uses the Modified National Institute of Standards and Technology database (MNIST) with 10 classes and 6,000 instances per class [21].

2.2 Conventional Approaches And Comparison

A comparison of publicly available frameworks is given. This is broken into two categories, data and model parallelism. Data parallelism is discussed first and followed by model parallelism. At the end of each subsection, the unique contribution of our work is discussed.

2.2.1 *Data Parallelism Comparison*

Data parallelism research has several similarities and distinctions between the various approaches. Research from Iandola et al. introduces FireCaffe [16]: used adjustable batch size to

reduce the total quantity of communications required during training. Communication was conducted using a reduction tree. When training GoogLeNet [11] and Network-in-network [34] on ImageNet [35] they achieved 36% and 30% speedup efficiency, respectively, on a 128 GPU cluster. Their strategy for scaling up DNN training was to focus on reducing communication overhead. They use fast interconnects such as Infiniband or Cray Gemini to accelerate communication among the GPUs.

Sergeev et al. introduced the Horovod [36] framework. Building on the realization a ring-allReduce approach can improve over reduction trees, motivated them to address Uber's TensorFlow needs. We adopted Baidu's draft implementation of the TensorFlow ring-allReduce algorithm and built upon it. NCCL 2 introduced the ability to run ring-allReduce across multiple machines, enabling them to take advantage of its many performance boosting optimizations.

Gu et al. built the cNeural [13] framework for fast training of large scale datasets with millions of training samples. To achieve this goal, firstly, cNeural adopts HBase for large scale training dataset storage and parallel loading. Secondly, it provides a parallel in-memory computing framework for fast iterative training. Third, they choose a compact, event-driven messaging communication model instead of the heartbeat polling model for instant messaging delivery.

Horovod, FireCaffe, cNeural, and our research all conduct data parallelism in nearly the same manner. The dataset is split amongst the nodes and each node has an exact copy of the model being trained. Each node trains the model using its specific data and the models are synchronized given a set frequency and respective communication method. Our research does not focus on improving the communication scheme of datasets with millions of instances, instead we focus on small scale implementation needs with lowered implementation complexity to the user. Our contribution is the automation module, it makes the parallelization invisible to the user

by providing them with a single function call that handles the data distribution, synchronization frequency, and setting the synchronization hooks during training.

Data parallelization benefits best from large datasets; however, due to our small-scale implementation and consistency in comparing data and model parallel results we restricted our testing to small size datasets. This does not fully demonstrate the capabilities of distributed neural network training with data parallelism and is discussed further in Section 4.2.3. We chose to allow this as our focus is on reducing implementation complexity.

2.2.2 *Model Parallelism Comparison*

We first described how the other libraries implement a model parallel method and drawbacks for these methods. Then we call out which libraries do not support model parallel training. Finally, we conclude by stating what our module does differently.

DistBelief [3], the predecessor to TensorFlow, is a framework outlined by Dean et al. The user defines the computation to take place at each node in each layer of the model, and the messages to be passed during the forward and backward phases of computation. For large models, the user may partition the model across several machines, so responsibility for the computation for different nodes is assigned to different machines. When a model is split across multiple nodes vertically, horizontally, or both, only the connections between the neurons on the edge of the graph are synchronized. This helps reduce the time needed for communication and allow flexibility in optimizing the network distribution for a specific topography. The downside is the user must define the computation to take place at each node in each layer, and the messages communicated during the forward and backward propagation. In Torch, Neural Networks are built using various layers and functions. We provided parallelized equivalents to these components. This allows the user to easily build their network just as they would normally

and choose which layers are parallelized and which are not. Enabling a high level of usability while still giving the user a large amount of control.

Commonly known from Google and outlined by Abadi et al., TensorFlow [5] is an interface for expressing machine learning algorithms, and an implementation for executing such algorithms. The system has been used for conducting research and for deploying machine learning systems into production across computer science and other fields, including speech recognition, computer vision, robotics, information retrieval, natural language processing, geographic information extraction, and computational drug discovery. They accomplish parallelism automatically by first analyzing the node topography. Given a computational graph, the input and output size, in bytes, of each layer is estimated, along with the computation time required. A simulated execution of the entire graph is then conducted, and the optimal device is chosen for each node using greedy heuristics. For model parallel training this means each node is given a layer of the network and the output and input between layers is communicated. The downside is parallelization is dependent on the number of layer present in the network. A three-layer network cannot be parallelized on more than three nodes.

Our parallelization is not configured using an elaborate method of estimation and heuristics, we simply divide each layer to be parallelized across the given nodes within the MPI handle. This allows parallelization to be implemented quickly but could pose a problem when a large network of compute nodes is used.

Horovod, FireCaffe, cNeural focus on data parallelism techniques and reducing communication overhead; therefore, there is no comparison between our model parallel approach and these libraries. We chose to implement automation for both data and model parallel approaches to avoid limiting the applicability and usefulness of our work. Model parallelism

works best on smaller datasets with complex neural networks. However, due to our small-scale implementation and consistency in comparing data and model parallel results we restricted our testing to small size models. This does not fully demonstrate the capabilities of distributed neural network training with model parallelism and is discussed further in Section 4.2.6. We chose to allow this, as our focus is on reducing implementation complexity.

Chapter 3. METHODS

This chapter outlines the key underlying systems chosen for our research, the considerations for each decision, benchmark configurations and tests, and datasets chosen. The high-level and detailed design for data and model automatic parallelization is analyzed. Metrics are clearly defined and explained why each is critical. Finally, key challenges and decisions for development are covered.

3.1 PLATFORMS FOR RESEARCH

This section describes all existing components used and experiment setup for our research. Benchmark tests show our setup is comparable to real world systems.

3.1.1 *Torch vs TensorFlow*

Two of the most common open source Machine Learning libraries are Torch [37] and TensorFlow [38]. Torch is a scientific computing framework with wide support for machine learning algorithms prioritizing GPU computation. It is easy to use and efficient, thanks to an easy and fast scripting language, LuaJIT, and an underlying C/CUDA implementation. TensorFlow is an open source software library for numerical computation using data flow graphs. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.

Many other libraries are available; however, we chose to analyze these two based on their popularity and ease of programming. To better determine which library is best for our application, we examine the underlying languages they utilize. TensorFlow uses python and Torch uses Lua and supports objects written in C. By examining the underlying languages, we

can analyze the inherited attributes present in each machine learning library; these points are shown in Table 3.3.

Table 3.3. Lua vs. Python by category [39]–[41]

Category	Python	Lua	Conclusion
Documentation	Excellent documentation filled with plain English describing functionality	Official Lua documentation is very thorough. There are also many online resources or books for beginners and advanced users.	Tied
Getting Started	Ships with a sizeable standard library, an IDE, and a text-based live interpreter.	Lua is small due to many of the components not being included in the core package.	Advantage Python
Syntax / ease-of-use	The lack of extra characters, like semicolons and curly braces, reduces distractions. The language uses natural English words such as 'and' and 'or', meaning beginners need to learn fewer obscure symbols. On top of this, Python's dynamic type system means code isn't cluttered with type information.	The Lua syntax is modeled from Modula, a language known for being a fantastic introduction to programming. The language uses natural English words such as 'and' and 'or', and dynamic type system means code isn't cluttered with type information.	Tied
Available Libraries	Is commonly used in data science and machine learning with many libraries for scientific computing, such as numpy, pandas, matplotlib, etc.	Lua supports many data science and machine learning applications; however, very few come with the initial package	Tied
Speed	Slower to run than Java	LuaJIT is nearly as fast as rival code in C.	Advantage Lua
Memory Management	Benchmarks using less memory are: k-nucleotide, reverse complement, and binary tree.	Benchmarks using less memory are: nbody, fasta, spectralnorm, and fannkuch-redux.	Advantage Lua

Cross-Platform support	Installs and works on every major operating system.	Lua can be built on any platform with a ANSI C compiler.	Tied
Variable usage	Built-in support and for types such as lists, dictionaries, and sets, as well as supporting features like foreach loops, map, and filter makes their use much easier.	Built Variables are declared as they are used on a global scope. This can cause subtle, difficult to find errors.	Advantage Python
Size	Python22.dll needs 824 Kb.	Lua interpreter with standard libraries needs 182 Kb.	Advantage Lua

Python and Lua show excellent performance benchmarks and have widespread support from both corporate and private contributors. Extensive add-on libraries are available for almost any type of scientific application. Python and Lua each have advantages, and no clear winner is decided without also evaluating the machine learning library.

Decision: When determining which library to use for our research, we chose Torch based on the following advantages points [39], [40], [42]:

- Footprint - TensorFlow is built on python whose basic library is 824 Kb vs Torch which is built on Lua and their basic library is 182 Kb. Python does have more modules available than Lua, but these relate mostly to visualization and this is not required for our application and would incur unnecessary overhead.
- Debugging - Torch is built for automatic differentiation (reverse-mode) while TensorFlow uses symbolic computation. This makes debugging painful as an error in the graph is harder to associate to a line in the code.
- Resource Utilization - Uses less memory on average and is already a popular choice for embedded applications.

- Customization - Fewer external modules allows bundling for specialized purposes

3.1.2 *Torch and TorchMPI*

While Torch was identified as the library of choice for our research, the standard Torch installation does not support inter-node communication. TorchMPI, a library released by the Facebook research team around the beginning of 2017, is an additional library providing many MPI communication paradigms as well as additional paradigms specific to ANN training. TorchMPI provides many functions and classes, the following are used for our research [19]:

- Basic functionalities allow starting, synchronizing and stopping processes.
  - **Start()** – starts MPI process on each node.
  - **Stop()** – stops MPI process on each node.
  - **Barrier()** – holds execution on each node until all nodes have reach the same barrier.
- Collectives wrap a subset of MPI collectives useful for deep learning. These collectives operate on Torch tensors and scalar values with synchronous or asynchronous flavors. Only synchronize calls are used.
  - **allreduceTensor()** – This function operates in the same manner as allReduce with MPI operation SUM. The input from each node is gathered and added together, every node is left with the exact summation result.
- NN extends Torch.nn with support for synchronous and asynchronous collectives to run on a distributed cluster of CPUs and GPUs. Only synchronize calls are used.
  - **synchronizeParameters()** – uses underlying allreduceTensor() functionality to ensure all models start with the same training parameters.

- **synchonizeGradients()** – uses underlying allreduceTensor() functionality to synchronize model gradient weights across all nodes during training.

3.1.3 *Hardware Setup – Cobra Cluster*

All development and experiments were conducted on the Cobra Cluster and built specifically for this research. The cluster consists of (4) Jetson TX1 developer boards, (1) 8-port 1gps switch, and OpenMPI 2.0.1. Initial analysis of the cluster was conducted using an MPI communication benchmark test and Wave2D simulation.

Table 3.4 show tests run on the Cobra Cluster and (4) UWB-320 machines for performance comparison. We can see the Time from Table 3.4 is slower for the TX1 cluster. This is not surprising as the TX1s are a low-powered embedded system. However, the Efficiency and Speedup when using 4 nodes with the Wave2D simulation is nearly identical between the two clusters. From Table 3.4, UWB-320 machines achieved an efficiency of 76.26% and the TX1s achieved an efficiency of 76.80%. Seeing the efficiency trend is nearly identical, we reasonably assume the efficiency will translate to the UWB-320 machines. This is important to provide proof to the general applicability of our results despite the embedded testing environment.

Table 3.4. Wave2D Simulation results using (4) TX1s and (4) UWB-320 machines. Time is the average from 10 test runs using a 576 x 576 matrix for 500 iterations.

Method	Machine	Time	Efficiency	Speedup
1 Node	TX1s	7,380,509	-	-
2 Nodes	TX1s	4,199,555	87.87%	1.76
3 Nodes	TX1s	3,013,061	81.65%	2.45
4 Nodes	TX1s	2,402,566	76.80%	3.07
1 Node	UWB	1,929,872	-	-
4 Nodes	UWB	632,621	76.26%	3.05

3.1.4 *Datasets*

CIFAR10 (Canadian Institute for Advanced Research 10): Krizhevsky's research originated the CIFAR-10 dataset. He started with the 80 million tiny images dataset [43] developed from researchers at MIT and NYU. They continued to assemble it by searching the web for images of every non-abstract English noun in the lexical database WordNet [44]. They used several search engines and kept the first 3000 results for each search term. They removed perfect duplicates and images in which an excessively large portion of the pixels were white, as they tended to be synthetic figures rather than natural images.

Krizhevsky paid students to label a subset of the tiny images dataset. The labeled subset consists of ten classes of objects with 6000 images in each class. The classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Duplicate images were removed from the dataset by comparing images with the L2 norm and using a rejection threshold liberal enough to capture not just perfect duplicates, but also re-saved JPEGs and the like. Finally, the dataset was divided into a training and test set, the training set received a randomly-selected 5000 images from each class. This dataset became known as the CIFAR-10 dataset [22], after the Canadian Institute for Advanced Research, which funded the project. The images are small, clearly labelled, and have no noise which makes the dataset ideal for this task with considerably much less pre-processing.

Recognizing an object's categories is fundamentally a problem of deformable shape matching [45]. In recent years, Convolutional Neural Networks (CNN) have made a breakthrough in the field by learning the advantages of high-level features of images [46]. CNNs naturally integrate low/mid/high level features and classifiers in an end-to-end multilayer fashion [47]. These high-level features from images, are elements such as the locations and orientations

of contours. The hope is such features would be much more useful than the raw pixels. The CNNs have some connection to the physical realities of human vision, as the visual cortex contains neurons to detect high-level features such as edges [22].

MNIST (Modified National Institute of Standards and Technology database): The Modified National Institute of Standards and Technology (MNIST) database originated from the NIST database and was created to eliminate the bias between Special Database (SD) 1 and SD-3. We use what is referred to as the regular database and the images are centered in a 28 x 28 image by computing the center of mass of the pixels and translating the image to position this point at the center of the field [21].

Isolated character recognition has been studied for the past 20 years and serves as an excellent base example for my parallelization research. According to Lecun et al [21], early research gradient-based learning performs better than other techniques. Convolutional neural networks are specifically designed to extract relevant features directly from pixel images.

3.1.5 *Networks*

Two different types of networks are used during the training, Convolutional Neural Networks (CNN) and Multi-Layer Perceptron (MLP) networks. CNNs used for CIFAR training are composed with a total of 11 layers, the first six layers are composed of two three-layer components which make up the feature extractor of the network. Each three-layer component has a spatial convolution layer, followed by a Rectified Linear Unit (ReLU) activation function, and finally a spatial max pooling layer. A view layer then a dropout layer separates the feature extractor from the classifier. The classifier is composed of the remaining 3 layers, two fully-connected layers separated by a ReLU. The main difference for MNIST training is the CNN uses Tanh instead of ReLU as an activation function. See Appendix B for complete network.

Second, MLPs used for MNIST training are composed of two or three fully-connected layers and labeled as MLP2 and MLP3 respectively. Each network starts with a Reshape layer to form the data. Then a Tanh activation function separates each full-connected layer. CIFAR data was not trained using MLPs.

3.1.6 *Training Method*

The standard Stochastic Gradient Descent (SGD) training method is used for evaluation of TorchAD-NN. The SGD algorithm is a drastic simplification over gradient descent. Instead of computing the gradient of each empirical risk exactly, each iteration estimates this gradient based on a single randomly picked example. The stochastic process depends on the examples randomly picked at each iteration. Since the stochastic algorithm does not need to remember which examples were visited during the previous iterations. In such a situation, the stochastic gradient descent directly optimizes the expected risk, since the examples are randomly drawn from the ground truth distribution [48].

3.2 SYSTEM DESIGN

This section described in detail our module design and gives an example of how each one operates. These automation modules are built on top of many existing layers, Figure 3.1, to provide extensive functionality with minimal complexity for the user. Our TorchAD-NN module is inserted between the user application on the top and all other layers on the bottom. It uses the functionality of TorchMPI and CUDA NN libraries while abstracting the complexity from the user.

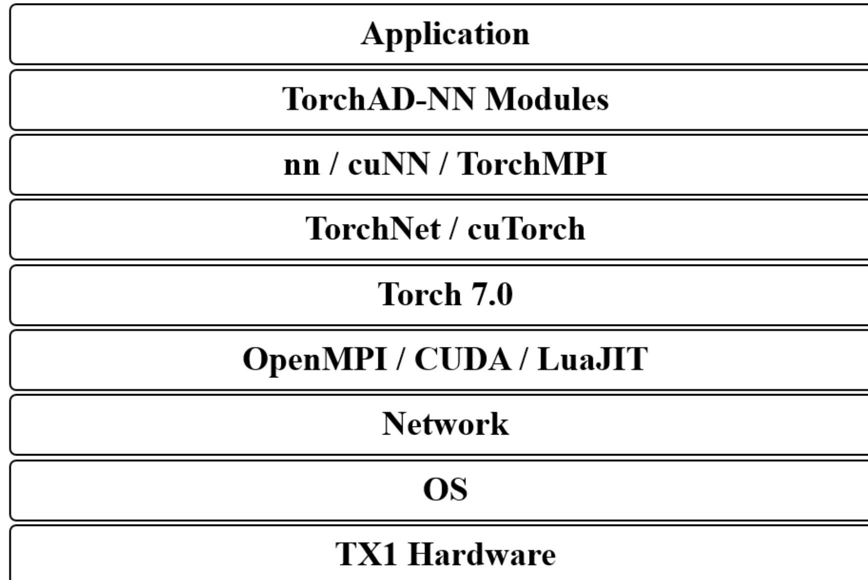

Figure 3.1. High level system architecture view

3.2.1 *Data Parallel Module*

We use Iandola’s common definition of *data parallelism* [16]. Defined as, each worker (i.e., TX1 node) gets an equal subset of the data, and then the workers communicate by exchanging weight gradient updates. For example, given our MNIST dataset with 60,000 samples and 4 TX1 nodes, each node gets 15,000 samples. Batch size determines synchronization frequency, at the end of each batch the network gradients are synchronized using synchronizeGradients() from TorchMPI. This function calls allreduceTensor() with MPI_OP sum, with model gradients as the input. At the end of training each node as a complete copy of the trained model. Figure 3.2 outlines this process.

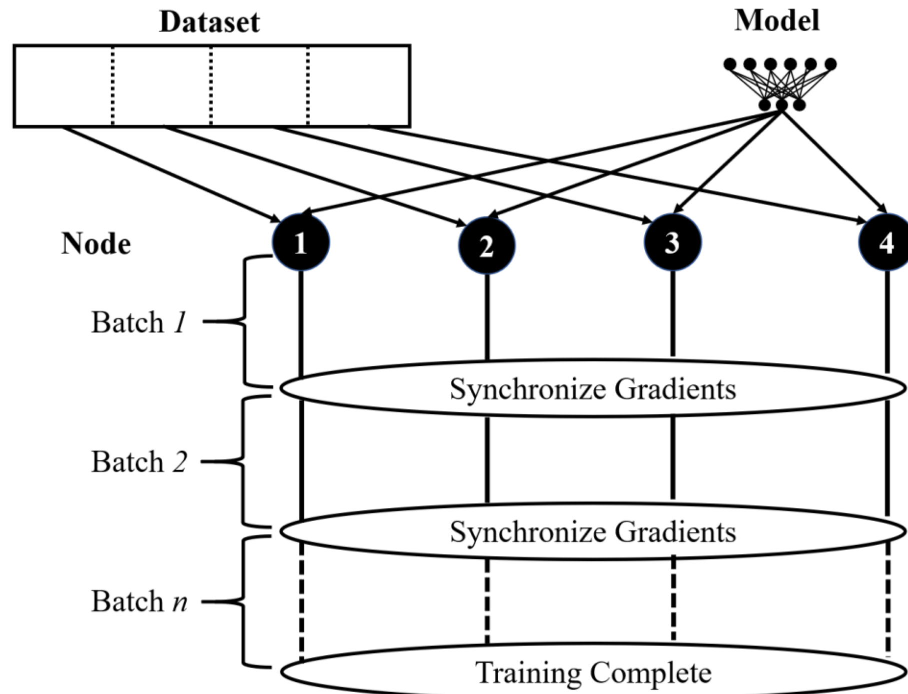

Figure 3.2. In data parallelism the data is partitioned equally across all nodes and each node gets a full copy of the network model. During training the gradients are synchronized at the end of each training batch. At the end of training each node contains a complete copy of the trained model.

The flowchart for TorchAD-NN data parallel is visualized in Figure 3.3. Automated parallelization can be broken into four stages:

1. Initialize MPI and MPINN handles
2. Split dataset on each node
3. Determine and set synchronization frequency
4. Override underlying training functions to perform model synchronization

Number (1) in Figure 3.3 shows the main function of the data parallel module called parallelize(). This function accepts the seven input values listed below. If no MPI and MPINN

object handles are passed in, the module will initialize both with GPU support, thus completing stage one. Following this Number (2) in Figure 3.3 is called.

- Input Values (* indicates required parameters)
  - *Dataset – data set to be split amongst nodes
  - *Targets – target labels for the given dataset
  - *Model – the ANN model used for training, this is used to access gradients for the model to be synchronized amongst nodes.
  - *Data Size – size of the given data set
  - MPI Object – handle for “torchmpi”, if one is not provided TorchAD-NN will start MPI communication
  - MPINN Object – handle for “torchmpi.nn” if one not is provided TorchAD-NN will start MPINN communication
  - Batch Size – represents the frequency of synchronization, if one is not provided TorchAD-NN will set frequency based on dataset size

Data_parallel() function, number (2) from Figure 3.3, is called twice, once with the dataset and once with the labels. Each time the data is split evenly amongst all the nodes present within the MPI communicator and assigned to a new data instance. If the dataset is unable to split evenly, any remainder will be dropped from the end of the dataset so each node has the exact same number of data instances. Once complete the new data instance is returned along with the new size. Once complete for both dataset and labels, this finishes stage 2 and number (3) from Figure 3.3 is called.

Optimize_sync() function, number (3) from Figure 3.3, performs the last two stages. If the user provided a batch size this value is used to set the synchronization frequency. If no value

is provided, one will be chosen using simple heuristics based on the size of the dataset. Communication speed analysis and synchronize frequency optimization functionality were removed from the scope of our research. It was decided measuring communication speed and applying this measurement to optimized synchronization is work highly dependent on the network and dataset. As our focus is to determine a general solution to improve usability, exerting effort to optimize synchronization doesn't directly benefit our focus. Setting the batch size finishes stage 3.

Number (4) from Figure 3.3, shows the two overridden functions necessary to allow automated synchronization. The function nn.StochasticGradient:train() allows training using the standard Stochastic Gradient Descent engine from Torch 7.0 to be parallelized and nn.Sequential:backward() allows any custom training using backward propagation to be parallelized. All basic functionality within these functions is unchanged, the only addition is the inclusion of a counter that when the synchronization frequency is met each node will call the synchronizeGradients() function. SynchonizeGradients() is provided by TorchMPI and described in Section 3.1.2. This concludes stage 4. Finally, the main function returns the three values listed below.

- Return Values
    - Data set – the new split dataset for the node
    - Targets – the new spit targets for the dataset
    - Data Size – the new dataset size

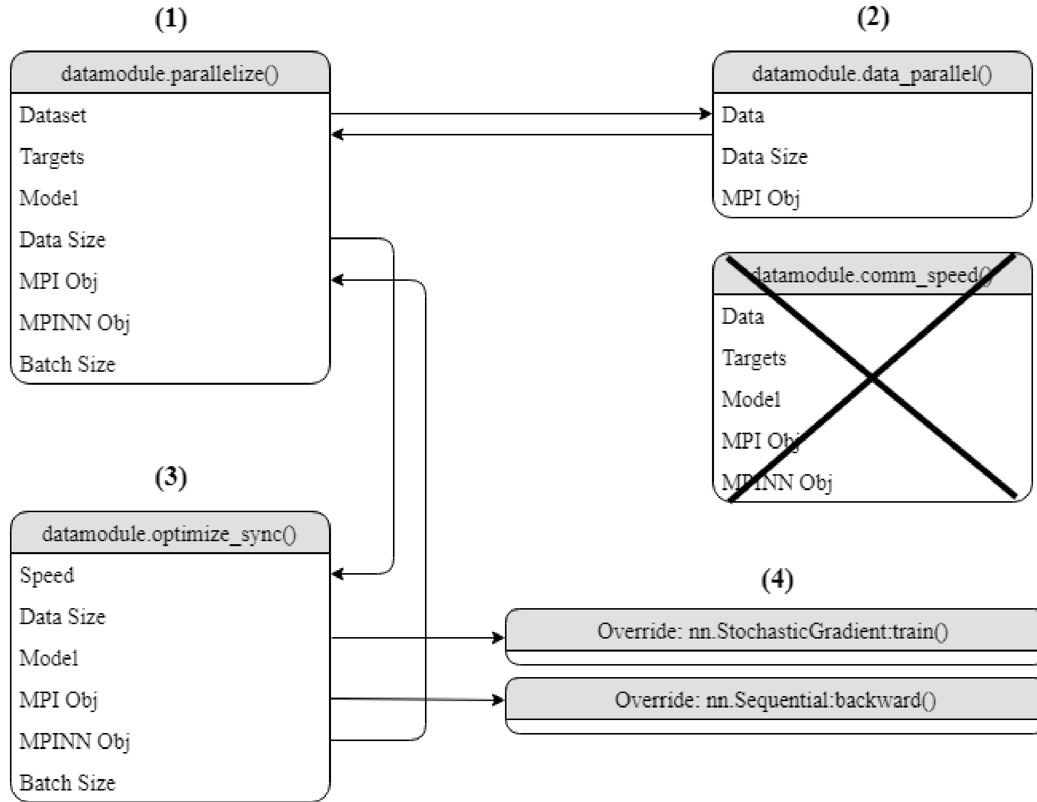

Figure 3.3. Flow chart outlining the four stages of automated data parallelization

3.2.2 *Model Parallel Module*

For *model parallelism*, we again use Iandol's definition [16]. Defined as, the case where each worker gets a subset of the model parameters, and the workers communicate by using allReduce to synchronize the output of a parallelized layer.

Our model parallelism is accomplished by splitting each layer across available nodes. For a single fully-connected layer with 1024 inputs and 10 output classes split across two nodes, each node takes 512 (1024 inputs / 2 nodes) inputs and each generates output size 10, output is synchronized across all nodes using allreduceTensor() and MPI_OP sum, Figure 3.4. This allows each node to share half of the input load and with correct synchronization minimizes accuracy loss. This method was inspired by Zhang et al. and their work with TorchMPI [49], [50].

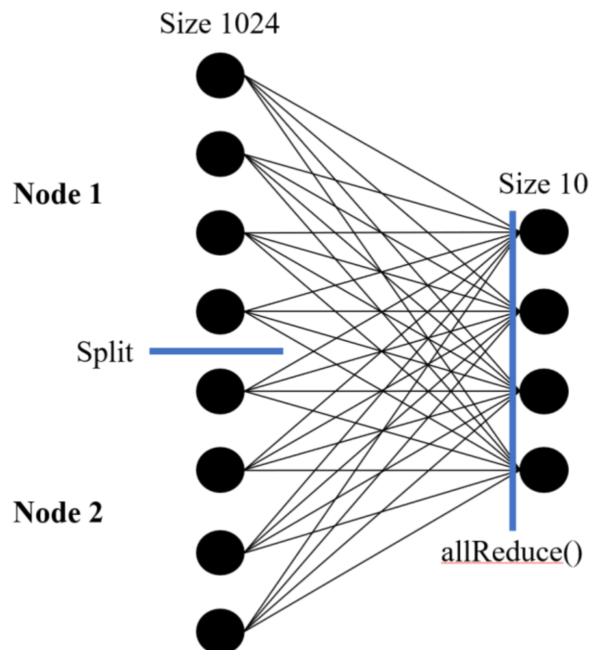

Figure 3.4. This model demonstrates input of size 1024 split between two nodes, each node takes half (512 of the inputs). Each node generates its own output of size 10 and is then synchronized between the nodes using allReduce.

Implementing this for a multi-layer network requires a deeper understanding of how the training takes places. For Torch's Stochastic Gradient training engine, a network trains through two phases, forward and backward propagation. Consider a MultiLayer Perceptron (MLP) network composed of two layers with the following components: linear1->tanh->linear2. During forward propagation the data moves through the network by entering linear1, then moving to tanh, and then entering linear2. For backward propagation the output from linear2 is then sent back through the network by entering linear2, then tanh, and finally linear1 in reverse. As a result, multilayer parallelization is more difficult as the output and input dimensions must be coordinated through both propagation phases.

Figure 3.5 illustrates the above example. It shows the narrowing and expansion of inputs and outputs as the model is trained through forward and backward propagation.

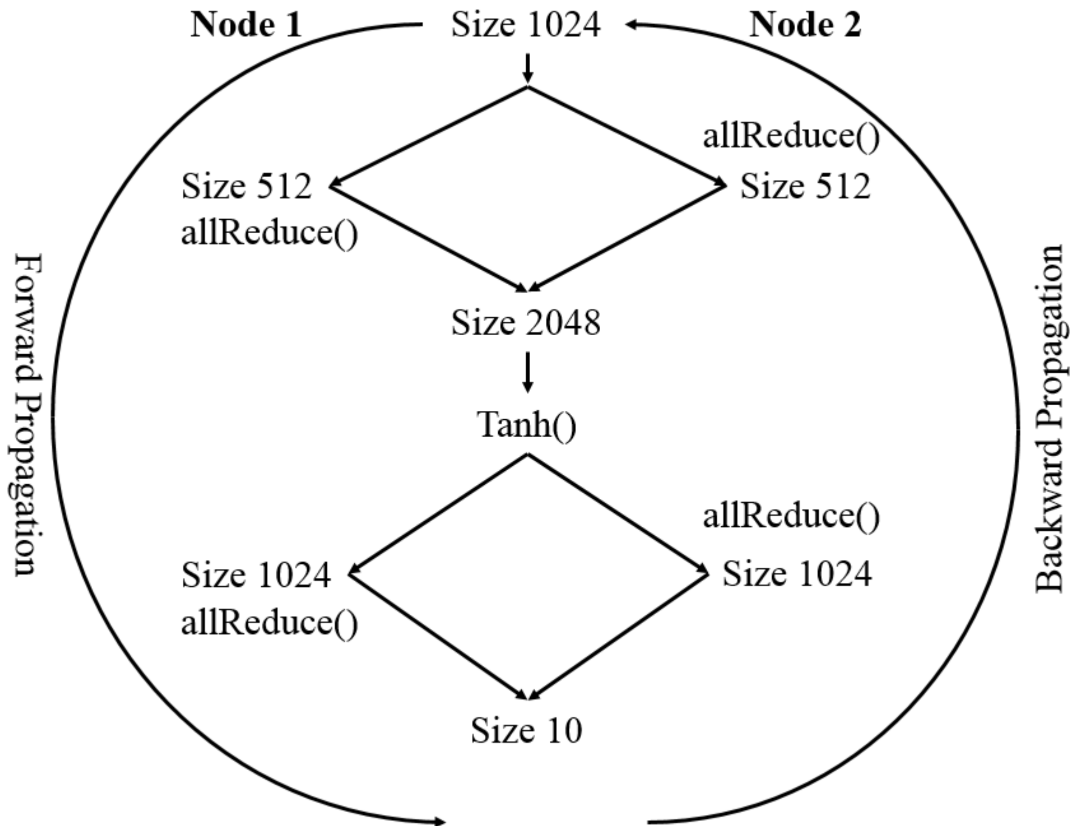

Figure 3.5. This diagram demonstrates the splitting and synchronization during forward and backward propagation using MLP2. Input starts as 1024, is narrowed to 512 between 2 nodes, and each generates their respective 2048 output. Each node then uses allReduce() to synchronize their outputs and proceeds through the Tanh component. After Tanh, the input is again narrowed from 2048 to 1024 on each node. AllReduce() is used again to synchronize each nodes output of size 10. This concludes the forward propagation. Backward propagation is the reverse, the output of 10 is then updated back to 1024, this 1024 is synchronized between the nodes with allReduce, which is fed into the Tanh layer. Here the Tanh layer must expand the 1024 to 2048 since the next layer is expecting this size output. The 2048 is then updated back up into each nodes 512 and again synchronized using allReduce(). Each full cycle of forward and backward propagation requires 4 allReduce() calls.

This parallelization is implemented on each component using a wrapper function. For the forward propagation phase, the function updateOutput() is wrapped for each component, and during backward propagation, the function updateGradOutput() is wrapped for each component. Table 3.5 shows the code from MPInitialLinear.updateOutput() as an example of how each component is wrapped. Line 2, shows the function input being narrowed using our custom narrowInput() function. This function evenly splits the input neurons across all nodes in the MPI communicator. Line 3, shows the unmodified updateOutput() function called with the narrowed input. Line 4, shows the output being synchronized using allReduce(). In this way, any changes to the underlying neural network libraries will be reflected in TorchAD-NN model parallelism.

Table 3.5. MPInitialLinear.updateOuput() source code

Line	**Code**
1	function MPInitialLinear.updateOutput(self, input)
2	local input = narrowInput(input)
3	self.output = nn.Linear.updateOutput(self, input)
4	mpi.allreduceTensor(self.output)
5	return self.output
6	end

Linear and Reshape components vary slightly depending on if they are at the top of a neural network or on any layer below. Initial Reshape and Linear are used if they are the first or second layer, otherwise Base Reshape and Linear are used. Three different operations are applied to variables within the components. Narrowed, is where values are evenly split amongst all nodes within the MPI communicator. Expanded, is where values are concatenated to reform their pre-narrowing size. Synchronized, is where the values are communicated across all nodes and each node is left with identical values. The list below shows each component wrapped, which functions are modified, and which operations are used.

Model Parallel Components

- MPInitialReshape – This component narrows the input when Reshape updateGradInput() is used at the top of the network. This corresponds to the narrowed output from the below linear layer's updateGradInput().
    - updateOutput() – unchanged
    - updateGradOutput()
        - Narrowed – input
- MPBaseReshape - Input for updateGradInput() is expanded for the above linear layer during backward propagation.
    - updateOutput() – unchanged
    - updateGradOutput()
        - Expanded - input
- MPInitialLinear – This component overrides the original and allows the input layer to be narrowed over multiple nodes within the network. Output is then synchronized using allreduceTensor(). For the initial linear layer accGradParameters() input must also be narrowed.
    - updateOutput()
        - Narrowed – input
        - Synchronized - output
    - updateGradOuptut()
        - Narrowed – input
        - Synchronized – output
    - accGradParameters()

        - Narrowed - input

- MPBaseLinear – This component overrides the original and allows the input layer to be narrowed over multiple nodes within the network. Output is then synchronized using allreduceTensor(). For base linear layer, accGradParameters() input is not narrowed.
    - updateOutput()
        - Narrowed – input
        - Synchronized - output
    - updateGradOuptut()
        - Narrowed – input
        - Synchronized – output
- MPTanh – This component narrows the input for updateGradInput() during backward propagation. Output from updateGradInput() is expanded for the above linear layer during backward propagation.
    - updateOutput() - unchanged
    - updateGradOuptut()
        - Narrowed – input
        - Expanded - output

3.2.3 *Development Lifecycle*

We used an Agile approach to the development of our automation modules. Sprints were done in 3-week increments. At the end of each Sprint, the current module would be presented to our committee chair as the deliverable along with a report. Status, lessons learned, and next steps were all assessed and then the next Sprint began. This methodology allowed us to react quickly

to new knowledge or challenges encountered. Additionally, we were able to quickly identify and adjust when we deviated from our timeline.

3.3 METRICS

The most critical metric used in our research is our measure of implementation complexity or usability. This metric is clearly defined and justified why it is an appropriate measure. Our performance metrics are also given in detail.

3.3.1 *Usability (Halstead's Complexity Measures)*

The original proposed complexity formula (3.1) is defined as the number of Lines of Code (LoC) multiplied by the number of Function Calls (FC) multiplied by the number Operands (O) needed for parallel implementation. If any of those three are a zero it is replaced by a one so the complexity score does not equal zero.

$$Complexity = LoC \times FC \times O \quad (3.1)$$

However, this complexity formula was only used for our initial analysis. To avoid the bias of surveying real people to assess usability, we needed a well-established metric. Two of the most common methods are McCabe's Cyclomatic complexity [51] and Halstead's software science [52].

Cyclomatic Complexity (CC) was first introduced by Thomas J. McCabe Sr. in 1976. Since then, it has become one of the most common methods to quantify the complexity of a program in a single number. CC is calculated by using the control flow graph of program and describes the non-linearity of this graph. In summary, it counts the number of linearly independent paths through the source code [51].

Halstead's Complexity Measures (HCM) were first introduced by Maurice Howard Halstead in 1977. Much like McCabe's metric, HCM has become one of the most commonly used systems to quantify a program's complexity. HCM goes even further, measuring various metrics, such as time to program, number of expected bugs, etc., see Table 3.7 for all metrics.

We chose to use HCM as our usability metric. CC only represents a program's complexity from the viewpoint of the computation to be performed and does not consider complexity from the viewpoint of the programmer. HCM delivers both the effort and time needed by the programmer to implement a given solution. Consequently, HCM proved to be the best available metric to accurately represent usability.

Operators are defined as keywords and symbols used to define a function. Examples include a function call, an if then statement, parentheses, semicolons, require keyword, etc.

Operands are defined as any variable or number used in the calculations. It can also refer to more complex entities such as the network model or keywords used to access a data instance, but not a function of these. See Table 3.6 for symbols and descriptions.

Table 3.6. HCM uses the following parameters:

Symbol	**Description**
n_1	Number of distinct operators
n_2	Number of distinct operands
N_1	Total number of operators
N_2	Total number of operands

Table 3.7. HCM provides the following metrics [53]*:*

Meaning	**Symbol**	**Formula**
Size of vocabulary	n	$n_1 + n_2$
Program length	N	$N_1 + N_2$
Program volume	V	N * log2 n
Difficulty level	D	$n_1/2 * N_2/n_2$
Effort to implement	E	V * D
Time to implement	T	E / 18
Number of delivered Bugs (Unused)	B	V / 3000

3.3.2 *Metrics for Performance Evaluation*

Speed – This is measured using the built-in system tic and toc functions. Time needed to test the trained model falls outside the scope of our research.

- Pre-processing time is measured from when the script starts to right before the network training function is called.
- Training time only measures the amount of time needed to train the model.

Speedup – Speedup is calculated by taking the time (either pre-processing or training) from a (1) node implementation and dividing it by the time from a (n) node implementation, see Equation 3.2. It represents how many times faster the program ran.

- SU-P: Speedup for Pre-processing time
- SU-T: Speedup for Training time

$$Speedup = \frac{Time_{single\ node}}{Time_{n\ nodes}} \qquad (3.2)$$

Efficiency – Efficiency is calculated by taking the speedup and dividing it by the number of nodes used for the faster implementation, see Equation 3.3. It is given as a percentage.

$$Efficiency = \frac{Speedup}{n\ nodes} \tag{3.3}$$

Accuracy – Accuracy is measured using a test data set and is calculated for each category using a confusion matrix and then aggregated to give the overall global accuracy of the trained model.

Accuracy Loss – the amount of accuracy lost when parallelizing the training, see Equation 3.4.

$$Accuracy\ Loss = Accuracy_{single} - Accuracy_{n\ nodes} \tag{3.4}$$

3.4 CHALLENGES

Although many challenges were faced during our development, the two most influential are outlined below.

3.4.1 *Synchronization*

A critical part of the automated data parallelization module is synchronizing between nodes in the network. This synchronization is conducted at the end of each batch of training. Three proposed solutions to this problem are:

1. User manually calls a synchronize function after backward propagation of the network.
2. Debug hooks catch every function call and determine if the call made is the backward function.
3. We write our own hook functionality.

Solution number 1, would be the easiest for us to implement. However, requiring the user to make additional function calls does not support our primary goal of improving usability.

Solution number 2, does not require anything additional from the user, but catching and comparing every function call creates a large amount of overhead. This overhead directly counters our secondary goal of increasing efficiency and thus this approach was not taken. The last option requires more work effort to implement but does not counter either of our goals.

Solution 3 was implemented by overriding two functions from the NN library, nn.Sequential:backward() and stochasticGradient:train(). This allows support for user customized training with forward()/backward() and SGD engine training.

3.4.2 *Spatial Convolution Layer Parallelism*

For a fully-connected linear layer, the neural nodes can be split across multiple machines because the input size and output size of the layer are independent. When the input size of 1024 narrowed across 2 nodes to an input size of 512 for each node, each node still has an output size of 2048.

Spatial convolution layers are used for building Convolutional Neural Networks (CNN) and tend to specialize in processing visual analysis. On convolution layers the input size and output size are dependent on each other, see Table 3.8. When the input size is split for parallelization the output size is also changed. Furthermore, the change is not directly proportional as other parameters play a role in determining output size. For example, unparallelized input and output on a convolution layer are 1x32x32 and 32x28x28 respectively. When the same input is split across 2 nodes, the new size is 1x32x16 (or half), but the new output size is 32x28x12. The new output is less than half of the original output.

Consequently, we were unable to develop a generic model parallel solution based on our narrowing and expansion techniques for these Spatial Convolution components. This limits the model parallelization to only networks built using fully-connected linear layers. Future work

could discover a method to parallelize these layers individually and allow for increased customization support from the automation module.

Table 3.8. Code snippet from SpatialConvolutionMM_updateOutput() [54] *demonstrating outputHeight and outputWidth are dependent on multiple parameters and do not scale proportionally with inputHeight and inputWidth.*

Line of Code
long inputHeight = input->size[dimh];
long inputWidth = input->size[dimw];
long outputHeight = (inputHeight + 2*padH - kH) / dH + 1;
long outputWidth = (inputWidth + 2*padW - kW) / dW + 1;

How to parallelize convolution layers in a CNN is a challenge other research has faced. Iandola et al. in their research with FireCaffe state how convolution layers, where the spatial resolution of the filters is smaller than the resolution of the activations, do better with data parallelism because it requires less communication [16]. Their conclusion was only models with large fully-connected layers benefited from model parallelism.

Keuper and Preundt published research regarding the theoretical and practical limits of DNN scalability. Parallelization of DNN training requires the communication of the computed gradients between all nodes in every iteration. And the entire communication must be completed before the next iteration. Naturally, one would try to overlap this communication (which can be done when split layer by layer) with the compute times. However, our layer split technique does not allow for communication overlap as, the conclusion of each layer must be communicated. The main pitfall to this strategy is the compute times per iteration are rather low and are decreasing further when scaling to more nodes. Ironically, faster compute units (i.e., GPUs) in increase the fundamental problem, the communication time exceeds the compute time [9].

Chapter 4. EVALUATION

This chapter outlines all details and parameters of each experiment. Main takeaways are discussed in Section 4.2 using a consistent format and supporting data provided for each topic.

4.1 Experiment Setup

Experiments were broken into two categories: Data Parallel and Model Parallel experiments. The specific datasets, variations, and parameters used are outlined in the following two subsections.

Parallel variation, refers to three methods of implementing parallelization. Single node is when the training is conducted on a single node using either sequential CPU execution or GPU parallel execution. Manual parallelization is when the training code from single node execution is converted to a manually coded multi-node solution. Automatic parallelization is when the training code from single node execution is converted automatically to a multi-node solution. Automatic parallelization is executed using the result of our research, TorchAD-NN.

4.1.1 *Data Parallel Setup*

Data parallel experiments were executed using the following list of properties. Complete results for all experiments is found in Appendix C.

- Hardware: four NVIDIA Jetson TX1 developer boards clustered using a single 1 gps switch, see Section 3.1.3.
- Dataset: CIFAR and MNIST, see Section 3.1.4.
- Dataset Size: MNIST training uses the full 60K instances. CIFAR training uses only 32K of the available 50K, due to device memory constraints.

- Model: Two variations of CNNs (one for each dataset), see Section 3.1.5.
- Training Methods: custom training function (use syncGradients after backward propagation) and SGD training engine.
- Parallel Variations: single node only, manual parallelization, and automatic parallelization module.
- CPU/GPU: MNIST training is done on CPU. CIFAR training is done on GPU.
- Dynamic Parameters: number of nodes and batch size.

4.1.2 *Model Parallel Setup*

Model parallel experiments were executed using the following list of properties. Complete results for all experiments is found in Appendix D.

- Hardware: four NVIDIA Jetson TX1 developer boards clustered using a single 1 gbps switch, see Section 3.1.3.
- Dataset: MNIST dataset, see Section 3.1.4.
- Dataset Size: 10K data instances used for CPU training, 2K data instances used for GPU training (readv error prevents more than 2K being used for training).
- Model: MLP2, MLP3, and CNN, see Section 3.1.5.
- Training Methods: All training with SGD engine.
- Parallel Variations: single node and automatic parallelization module. Three synchronization variations on MLP3: all three layers, first and last layer, and only first layer.
- CPU/GPU: both.
- Dynamic Parameters: Fully-connected layer size.

4.1.3 *Complexity Evaluation Example*

Two examples using HCM are given below. Example one, is using the CIFAR dataset with a custom training algorithm. Table 4.9 shows the added lines of code for manual parallelization. On the right of the table the number of operators and operands is given for each line of code. At the bottom of the table these are summed to get the total operators and operands from Table 3.6. In the same manner, Table 4.10 demonstrates HCM using automated parallelization. Example two, is using the CIFAR dataset with SGD training. Table 4.11 shows the added lines of code for manual parallelization. On the right of the table the number of operators and operands is given for each line of code. At the bottom of the table these are summed to get the total operators and operands from Table 3.6. In the same manner, Table 4.12 demonstrates HCM using automated parallelization.

Table 4.9. CIFAR Backward() Manual Parallelization Comparison

Line	**Code**	**Operators**	**Operands**
24	local mpi = require('torchmpi')	5	2
25	mpi.start(true)	3	2
263	local mpinn = require('torchmpi.nn')	6	3
264	mpinn.synchronizeParameters(model)	3	2
266	stripe = trsize / mpi.size()	5	3
293	for t = (mpi.rank() * stripe) + 1,dataset:size() - ((mpi.size() - (mpi.rank() + 1)) * stripe),opt.batchSize do	28	11
334	if i == opt.batchSize then	4	3
336	mpinn.synchronizeGradients(model)	3	2
337	end	1	0
383	end	1	0
473	mpi.barrier()	3	1
551	mpi.stop()	3	1
Totals		n1 = 26	n2 = 14
Totals		N1 = 65	N2 = 30

Table 4.10. CIFAR Backward() Auto Parallelization Comparison

Line	Code	Operators	Operands
22	automation = require 'automatedparallelization'	3	2
184	trainData.data, trainData.labels, trsize = automation.datamodule.parallelize(trainData.data, trainData.labels, model, trsize, nil, nil, opt.batchSize)	18	17
Totals		n1 = 7	n2 = 11
Totals		N1 = 21	N2 = 19

Table 4.11. CIFAR SGD Train() Manual Parallelization Comparison

Line	Code	Operators	Operands
25	local mpi = require('torchmpi')	5	2
26	mpi.start(true)	3	2
175	local mpinn = require('torchmpi.nn')	6	3
176	mpinn.synchronizeParameters(model)	3	2
178	stripe = trsize / mpi.size()	5	3
183	newdata = {	2	1
184	data = {},	3	1
185	labels = {},	3	1
186	size = function() return stripe end }	5	2
189	local start = (mpi.rank() * stripe) + 1	8	4
190	local finish = start + (stripe - 1)	5	4
191	newdata.data = trainData.data[{ {start,finish} }]	7	6
192	newdata.labels = trainData.labels[{ {start,finish} }]	7	6
203	if (self.sync_counter == nil) then	5	2
204	self.sync_counter = 1	3	2
205	end	1	0
208	if (self.sync_counter == stripe) then	6	2
209	mpinn.synchronizeGradients(model)	3	2
210	elseif (self.sync_counter % opt.batchSize == 0) then	8	4
211	mpinn.synchronizeGradients(model)	3	2
212	end	1	0
213	self.sync_counter = self.sync_counter + 1	6	3
305	mpi.barrier()	3	1
350	mpi.stop()	3	1
Totals		n1 = 29	n2 = 21
Totals		N1 = 104	N2 = 56

Table 4.12. CIFAR SGD Train() Auto Parallelization Comparison

Line	Code	Operators	Operands
23	automation = require 'automatedparallelization'	3	2
172	trainData.data, trainData.labels, trsize = automation.datamodule.parallelize(trainData.data, trainData.labels, model, trsize, nil, nil, opt.batchSize)	18	17
Totals		n1 = 7	n2 = 11
Totals		N1 = 21	N2 = 19

4.2 MAIN TAKEAWAYS

This section discusses the main observations from TorchAD-NN experiments. Each topic is introduced separately and analyzed in the following manner: (A) introduces the main idea, (B) explains the relevant data, (C) analyzes why the observation occurred, (D) discusses any negative associated with this observation, and (E) delivers the conclusion and impact of this observation.

Please Note:

- Single node implementations are given a grey background for ease of identification.
- Line reference numbers are given corresponding to the complete test entry in appendices.
- Train() refers to training using StochasticGradient:train()
- Backward() refers to any custom training function which utilizes model:backward().

4.2.1 *Data Parallel Complexity (Usability)*

A. **Main Topic:** Halstead's measures (HCM) show TorchAD-NN Data Parallelism requires significantly less effort to implement than manually coding a parallel solution.

B. **Supporting Data:** Table 4.13 records the calculated effort and time needed for each implementation variation. These numbers show TorchAD-NN is nearly 14x easier to implement on custom training and 22.5x easier for SGD Training. GPU values are given as a baseline, since inspiration for the modules came from the ease of porting ANN training to the GPU. An improvement was also achieved from the GPU baseline.

C. **Why this happens:** TorchAD-NN has a low effort because it encapsulates nearly all the necessary parallelization techniques and reduces them to a single function call. Although, this does not decrease the overall code complexity, it does make it invisible to the user. The user can access the techniques for data parallelism with a single function call. Manual complexity is high because it requires only subsections of the dataset to be iterated over on each node, and this subset must be calculated specific to its respective rank ensuring the entire dataset is covered. These calculations have a high effort associated with them.

D. **Drawbacks/Limitations:** If the user has a non-standard data representation, this would cause issues with the data parallel module. Because the parallelization is hidden from the user, if there was an error with the data format, it could be difficult to track down the underlying issue. This is mitigated by testing on two common datasets and assuming the data is in the Torch Tensor representation. Another drawback, TorchAD-NN does not currently support feed-forward only network training. This only affects the synchronization part of the module and not the data splitting. To overcome this limitation an additional line of code to call the synchronization function needs to be added, raising the effort only slightly.

E. **Conclusion/Summary:** By encapsulating the necessary operations to hide data splitting and synchronization from the user to a single function call, TorchAD-NN significantly reduces coding effort, which increases usability. Supporting backward synchronization allows a broader range of support to custom training functions, and with an additional line of code different training such as feed-forward only is also supported.

Table 4.13. Effort and Time measurements for data parallel implementations using HCM.

Implementation	Effort (HCM)	Time (HCM)
CIFAR		
Manual Backward()	14,085.73	782.5 seconds
GPU Backward()	5,277.49	293.2 seconds
Automated Backward()	1,008.31	56.0 seconds
Manual SGD Train()	22,695.79	1,260.9 seconds
GPU SGD Train()	1,423.01	79.1 seconds
Automated SGD Train()	1,008.31	56.0 seconds

4.2.2 *Data Parallel Performance*

A. **Main Topic:** MNIST and CIFAR results show TorchAD-NN is as efficient or more efficient than manual parallelization.

B. **Supporting Data:** Table 4.14 demonstrates for both Train() and Backward() executed on the CPU with MNIST the average speedup efficiency is 99.25% with only an average model accuracy loss of 0.93%. Table 4.15 demonstrates for both Train() and Backward() executed on the GPU with CIFAR the average speedup efficiency is 79.69% with an average model accuracy loss of 5.34%.

C. **Why this happens:** CIFAR training is conducted with five iterations and MNIST with one iteration. Therefore, CIFAR's 5 iterations times 32K data samples equals 160K training iterations compared to MNIST's 1 iteration times 60K data samples equals 60K training iterations. This means the calculation cost to communication overhead ratio is higher for MNIST training because it is being trained on the less efficient CPU. A higher calculation cost allows the communication overhead to be hidden and therefore the speedup efficiency for MNIST CPU is nearly 100%. Because CIFAR training is conducted on the GPU the calculation cost is lower and the communication overhead is more difficult to hide. Therefore, speedup efficiency on CIFAR GPU is less efficient than MNIST CPU.

   The simplicity and ease of training attributed to MNIST is why there is a dramatic difference in accuracy loss between MNIST and CIFAR training. See Section 4.2.3 for more detail.

D. **Drawbacks/Limitations:** CPU shows higher parallelization efficiency; however, the performance of CPU training compared to GPU training makes it an unwise choice. Using

the more complex CIFAR dataset, one iteration takes 106 seconds and on the other side one iteration with MNIST (s simpler dataset) takes 174 seconds on the CPU.

E. **Conclusion/Summary:** MNIST training on the CPU shows higher efficiency and lower accuracy loss. This is attributed to the simplicity of the dataset and the high calculation cost of the CPU. GPU training and more complex datasets would better demonstrate the global applicability of TorchAD-NN advantages.

Table 4.14. MNIST CPU Data Parallel Parallelization Performance.

MNIST – Data Parallel						
Appx. Ref.	**Nodes**	**Batch Size**	**Train (sec)**	**Accuracy**	**Acc-Loss**	**Efficiency**
Single CPU SGD Train()						
C.36	1	10,000	174.365	98.27%	-	-
Auto SGD Train()						
C.40	2	10,000	87.058	98.03%	-0.24%	100.14%
C.44	4	10,000	44.345	97.40%	-0.87%	98.30%
Manual SGD Train()						
C.48	2	10,000	86.385	98.03%	-0.24%	100.92%
C.52	4	10,000	46.797	97.40%	-0.87%	93.15%
Single CPU Backward()						
C.53	1	100	411.567	76.14%	-	-
Auto Backward()						
C.56	2	100	204.129	81.93%	5.79%	100.81%
C.59	4	100	102.029	67.86%	-8.28%	100.85%
Manual Backward()						
C.62	2	100	206.67	81.83%	5.69%	99.57%
C.65	4	100	102.861	67.69%	-8.45%	100.03%

Table 4.15. CIFAR GPU Data Parallel Parallelization Performance. Five training iterations.

CIFAR-10 – Data Parallel						
Appx. Ref.	**Nodes**	**Batch Size**	**Train (sec)**	**Accuracy**	**Acc-Loss**	**Efficiency**
Single GPU SGD Train()						
C.3	1	1,000	533.00	47.71%	-	-
Auto SGD Train()						
C.8	2	1,000	326.16	39.51%	-8.20%	81.71%
C.11	4	1,000	177.26	34.28%	-13.43%	75.17%
Manual SGD Train()						
C.14	2	1,000	315.93	40.99%	-6.72%	84.35%
C.17	4	1,000	187.64	34.36%	-13.35%	71.01%
Single GPU Backward()						
C.20	1	100	185.50	10.09%	-	-
Auto Backward()						
C.23	2	100	106.03	9.75%	-0.34%	87.48%
C.26	4	100	61.30	9.86%	-0.23%	75.65%
Manual Backward()						
C.29	2	100	107.51	9.84%	-0.25%	86.27%
C.32	4	100	61.13	9.92%	-0.17%	75.86%

4.2.3 *Data Parallel Unique Attribute (Batch Size)*

A. **Main Topic:** During data parallel training, varying batch size has a strong effect on training time, but only a minor effect on model accuracy when compared to the corresponding single node implementation.

B. **Supporting Data:** Table 4.16 shows model accuracy varies slightly when batch size is changed. Line D.9 – D.11 shows a change from 33.68% to 34.28% accuracy when batch size goes from 100 to 1,000. In contrast, for the same, lines 7-9, efficiency jumps from 31.02% to 75.17%. This comparison is illustrated in Figure 4.1. The accuracy, shown in Table 4.17, when parallelized to 2 or 4 nodes is comparable with using a single node and the dataset halved or quartered respectively.

C. **Why this happens:** For our training, batch size corresponded to synchronization frequency. The CIFAR dataset with 32K samples and a batch size of 100 needs to be synchronized 320 times versus a batch size of 1,000 only requires 32 synchronizations per training iteration. Because parallelization efficiency is dependent on hiding communication overhead, the fewer synchronizations the less overhead to be hidden. Therefore, we see a large increase in efficiency as batch size grows.

   Table 4.17, demonstrates the accuracy drop from 1 to 2 (Line D.4 and D.7) nodes and 1 to 4 (Line D.5 and D.10) nodes corresponds with a similar drop in accuracy if a single node is trained on either half or a quarter of the dataset respectively. This means most of the model on any given node is acquiring most of its accuracy from its dataset. The gradients being synchronized every 100, 500, or 1,000 data samples has a minimal positive impact on model accuracy. This same drop in accuracy trend is observed for manual and automatic implementations. The limitation is from the dataset used, not the TorchAD-NN module.

D. **Drawbacks/Limitations:** MNIST and CIFAR can be trained quickly on a single node with high accuracy. This means parallelization is relatively ineffective if a single node can accomplish the job in minutes. The performance results show the importance of using large datasets with data parallel training.

E. **Conclusion/Summary:** Synchronization incurs a large overhead when done frequently, and less frequent synchronization impacts the model accuracy. Large datasets too big for a single node are needed to demonstrate the advantages of this module.

Table 4.16. CIFAR GPU Data Parallel Parallelization Batch Size Comparison. One training iteration.

CIFAR-10						
Appx. Ref.	**Nodes**	**Batch Size**	**Train (sec)**	**Accuracy**	**Acc-Loss**	**Efficiency**
GPU SGD Train()						
C.1	1	100	547.18	48.57%	-	-
C.2	1	500	545.71	47.00%	-	-
C.3	1	1,000	533.00	47.71%	-	-
Auto SGD Train()						
C.6	2	100	559.81	39.98%	-8.59%	48.87%
C.7	2	500	333.97	41.40%	-5.60%	81.70%
C.8	2	1,000	326.16	39.51%	-8.20%	81.71%
C.9	4	100	441.06	33.68%	-14.89%	31.02%
C.10	4	500	202.89	33.32%	-13.68%	67.24%
C.11	4	1,000	177.26	34.28%	-13.43%	75.17%
Manual SGD Train()						
C.12	2	100	579.44	40.60%	-7.97%	47.22%
C.13	2	500	349.33	39.61%	-7.39%	78.11%
C.14	2	1,000	315.93	40.99%	-6.72%	84.35%
C.15	4	100	450.41	33.43%	-15.14%	30.37%
C.16	4	500	206.66	34.08%	-12.92%	66.02%
C.17	4	1,000	187.64	34.36%	-13.35%	71.01%

Table 4.17. CIFAR GPU Data Parallel Parallelization comparison between parallelization and reduced dataset on a single node.

CIFAR-10						
Appx. Ref.	**Nodes**	**Data Size**	**Train (sec)**	**Accuracy**	**Acc-Loss**	**Efficiency**
GPU SGD Train()						
C.4	1	16K	264.51	39.22%	-	-
C.5	1	8K	136.78	34.06%	-	-
Auto SGD Train()						
C.7	2	32K	333.97	41.40%	-5.60%	81.70%
C.10	4	32K	202.89	33.32%	-13.68%	67.24%

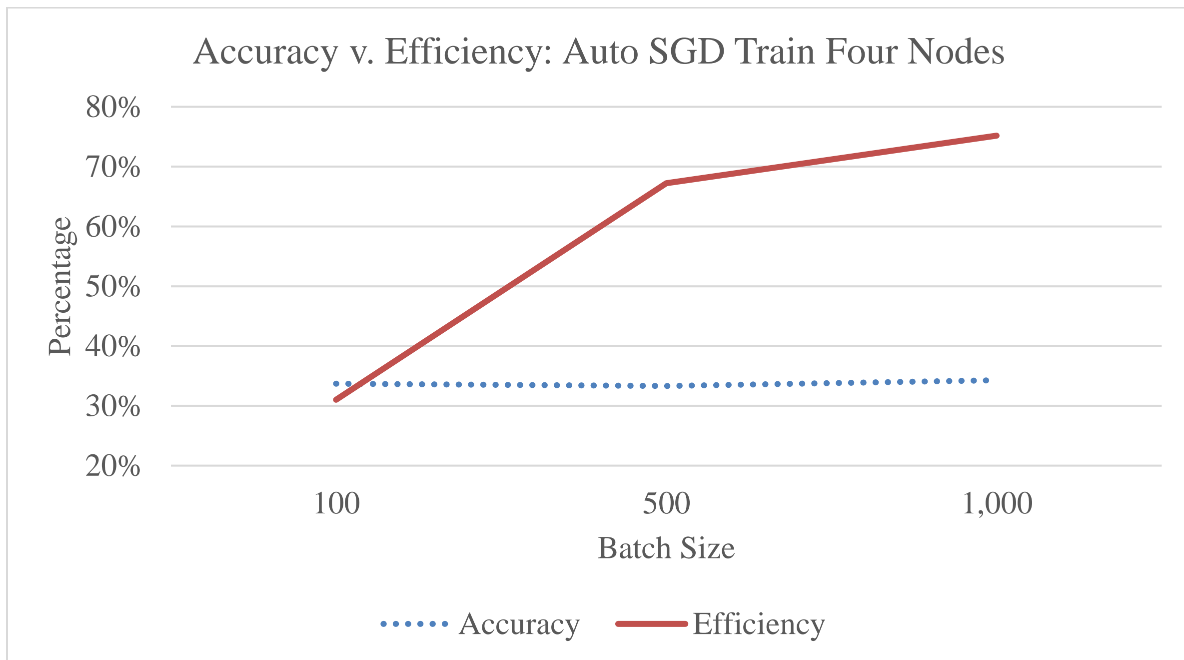

Figure 4.1. Data from lines D.9, D.10, and D.11 of Table 4.16, plotted to show the comparison between accuracy and efficiency when batch size is varied between 100, 500, and 1000.

4.2.4 *Data Parallel Unique Attribute (Pre-processing)*

A. **Main Topic:** TorchAD-NN demonstrates a strong increase in pre-processing efficiency when using custom training.

B. **Supporting Data:** Table 4.18 records pre-processing times and pre-processing speedup on MNIST dataset across 1, 2, and 4 nodes. Table 4.19 records pre-processing times and pre-processing speedup on CIFAR dataset across 1, 2, and 4 nodes. The single node pre-processing time required for CIFAR is approximately 37x more than time required for MNIST. Table 4.18 also demonstrates the pre-processing time required for 2 and 4 nodes is larger than the time required for 1 node.

C. **Why this happens:** First, it is necessary to look at why the pre-processing time increases for MNIST. When 2 or more nodes are used, the TorchAD-NN data parallel module is executed. This module does add a small amount of overhead as the dataset is split and synchronization hooks are set. When only a small amount of time, 1-2 seconds, is required for pre-processing the data parallel module overhead is more than the speedup gained by processing only a subsection of the dataset on each node. As a result, the speedup gained is overshadowed by the overhead and thus the overall pre-processing time required increases.

   When more pre-processing time is required, CIFAR requires 37x more time, the data parallel module overhead does not hide the speedup gain. Consequently, data parallel module on the CIFAR dataset achieves 1.84x speedup on 4 nodes.

D. **Drawbacks/Limitations:** Manual data parallel is not compared in this section because it does not alter the dataset on each node, but only iterates through a subsection. Therefore, no difference in pre-processing time required is observed. Although not represented in our complexity measure, it is important the user place the data parallel module function call after

the dataset is loaded but before pre-processing occurs. If data parallel module is executed after pre-processing no speedup is gained.

E. **Conclusion/Summary:** Increasing pre-processing efficiency was not a research goal; however, it is a natural improvement to be observed when less data is being pre-processed on each node.

Table 4.18. MNIST pre-processing time comparison across 1 to 4 nodes.

MNIST CPU Auto backward()			
Appx. Ref.	**Nodes**	**Pre-pro (sec)**	**SU-pre**
C.53	1	1.689	-
C.56	2	2.123	0.80
C.59	4	2.726	0.62

Table 4.19. CIFAR pre-processing time comparison across 1 to 4 nodes.

CIFAR-10 GPU Auto backward()			
Appx. Ref.	**Node**	**Pre-pro (sec)**	**SU-pre**
C.18	1	62.52	-
C.21	2	41.92	1.49
C.24	4	33.97	1.84

4.2.5 *Model Parallel Complexity (Usability)*

A. **Main Topic:** Lines of Code (LoC) required for implementation show TorchAD-NN Model Parallelism requires significantly less effort to implement than manually coding a parallel solution.

B. **Supporting Data:** Table 4.20 demonstrates the number of LoC required to implement each of the three variations. With only requiring 4 LoCs the TorchAD-NN model parallel module requires 33% fewer LoCs than GPU SGD Train() and 97% fewer LoCs than Manual Model Parallel SGD Train(). GPU values are given as a baseline, since inspiration for the modules came from the ease of porting ANN training to the GPU. We even improved usability from the GPU baseline.

C. **Why this happens:** HCM was not used here as the difference in implementation effort between manual and auto would have been significant. This significance would have detracted from the truth here. In data parallel, the code between manual and auto differs. Manual, only allows training to iterate over a section of the data, where Auto reforms the dataset. Synchronization is also handled differently: in automatic parallelization, functions are overridden, where in manual parallelization, a synchronization function is called after each training iteration. For data parallel, HCM accurately captured the difference in complexity.

   In contrast, model parallel auto and manual have identical code. For manual, the parallelized layer code is included as part of the training file. For auto, the parallelized layer code is included as part of the TorchAD-NN module and only requires the uses replace the component names (e.g., replace a nn.Linear call with nn.MPBaseLinear call). HCM would mislead into thinking there was a difference in implementation complexity because the code

was different. When the only difference is abstraction away from the user. To represent this, we chose to use LoC as a metric.

D. **Drawbacks/Limitations:** The standard Torch NN library contains many more components than included within the TorchAD-NN library. Each layer component requires analysis and specific parallelization approach, which also depends on where it falls within the network structure. This cost, led to only a limited number of layers being supported. This was to ensure each component could be well designed and tested. The most common layer components were chosen for parallelization. The key to successful parallelization is the ability to hide communication overhead with calculations. GPU calculation speed was too fast too successfully hide communication overhead, this is discussed further in Section 4.2.2.

   Users must have knowledge of which component to use depending on position within a network. For example, if Reshape is used as the first layer (as in the MLP2 and MLP3 networks) then MPInitialReshape() must be called. However, if it falls within the middle of the network (as in CNN) then MPBaseReshape() must be called. This extra cost on the user is not reflected in LoC and although minimal is still worth remembering to evaluate the whole picture.

E. **Conclusion/Summary:** By supporting individual component parallelization in TorchAD-NN, it brings the power of custom parallelized neural networks to the user with minimal complexity. This gives a high level of customization and flexible to how a model is parallelized.

Table 4.20. Lines of Code necessary for each variation of model parallelism.

Implementation	**Lines of Code**
GPU SGD Train()	6 Lines
Manual Model Parallel SGD Train()	131 Lines
Auto Model Parallel SGD Train()	4 Lines

4.2.6 *Model Parallel Performance*

The following two subsections outline the specific details for CPU and GPU implementations.

4.2.6.1 CPU

A. **Main Topic:** CPU Parallelization increases efficiency as the network increases in complexity.

B. **Supporting Data:** Table 4.21 shows as model complexity increases from 2048/2048 to 4096/4096 (size of second and third layers) speedup efficiency also increases for 2 and 4 node tests.

C. **Why this happens:** Computation cost versus communication overhead ratio is critical to effective parallelization. Increasing the size of each fully-connected layer greatly increases the computation cost. As the calculation cost increases, the communication overhead does not. As the ratio tilts toward computation overhead, efficiency increases.

D. **Drawbacks/Limitations:** This same principle holds true for GPU execution; however, as we discuss further in Section 4.2.6.2, a single GPU is significantly faster than even four CPUs.

E. **Conclusion/Summary:** Model parallelism across CPUs demonstrates increasing efficiency as the size of the model increases, yet the high speed of GPU training makes CPU less than optimal.

Table 4.21. MLP3 CPU model complexity SGD Train() compared with speedup efficiency trained on MNIST dataset.

CPU Model Complexity Tests						
Appx. Ref.	**Model Type**	**Nodes**	**SLS**	**TLS**	**Time (sec)**	**Efficiency**
D.71	MLP3	1	2048	2048	679.69	-
D.72	MLP3	2	2048	2048	418.13	81.28%
D.73	MLP3	4	2048	2048	300.04	56.63%
D.74	MLP3	1	4096	2048	1351.62	-
D.75	MLP3	2	4096	2048	764.77	88.37%
D.76	MLP3	4	4096	2048	483.04	69.95%
D.77	MLP3	1	4096	4096	2230.24	-
D.78	MLP3	2	4096	4096	1222.38	91.23%
D.79	MLP3	4	4096	4096	728.89	76.49%

4.2.6.2 GPU

A. **Main Topic:** GPU parallelization fails to exceed single node speed.

B. **Supporting Data:** From Table 4.22, it is seen all five model variations train slower on two GPUs versus one GPU. On this table, an efficiency of less than 50% indicates the training took longer than the single node implementation.

C. **Why this happens:** The CPU training from the previous section took hundreds of seconds to train a model even on the simpler MNIST dataset. When cut in half by splitting the training across two nodes, the communication overhead incurred is still significantly less than the training time required. Take line E.97 from Table 4.22 as an example. Assume by splitting the training across two nodes the time is halved (this assumes perfect parallelization for the sake of argument) from 9.64 seconds to 4.82 seconds, take the total time for two nodes minus the estimated time for training, 29.77 seconds minus 4.82 seconds, and this leaves an estimated 24.95 seconds of communication overhead incurred. Because the original time was only 9.64 seconds, there is no way parallelization can ever receive a speedup unless communication overhead is reduced, or computation cost is increased.

From Table 4.21, take the first line as an example, the CPU training time required is 679.69 seconds. Estimating perfect parallelization, halves the training time to 339.85, and adding communication overhead gives a total of 364.8 seconds. Because CPU training has a higher computation cost, the communication overhead is still hidden by the large decrease in training time. When considering GPU training, the communication overhead is the same, since the network and hardware have not changed, but now training takes less than 10 seconds. Therefore, we see a high efficiency with CPU parallelization and not with GPU parallelization.

Only one and two node results are shown, adding additional nodes only increases communication overhead, but does not address the ratio difficulty. The last two lines of Table 4.22 demonstrated CNNs can be trained in less time and with higher accuracy than any of the MLP3 models.

D. **Drawbacks/Limitations:** GPU memory restricted the number of training samples to 2K for these experiments. However, increasing the dataset size does not affect the ratio. As dataset size increases, the computation cost also increases, but the communication overhead also rises proportionally.

E. **Conclusion/Summary:** Increasing the time to train on each sample (e.g., increasing dataset complexity or model complexity) or reducing communication overhead (e.g., increase batch size to reduce communication frequency) are the only ways to increase GPU parallelization efficiency.

Table 4.22. MLP and CNN GPU SGD Train() speedup efficiency trained on MNIST dataset.

GPU Model Type Tests					
Appx. Ref.	**Model Type**	**Nodes**	**Time (sec)**	**Accuracy**	**Efficiency**
D.94	MLP2	1	5.87	67.90%	-
D.95	MLP2	2	20.93	72.50%	14.02%
D.97	MLP3	1	9.64	77.30%	-
D.98	MLP3	2	29.77	66.30%	16.19%
D.100	MLP3	1	9.64	77.30%	-
D.101	MLP3-half	2	22.04	69.30%	21.87%
D.103	MLP3	1	9.64	77.30%	-
D.104	MLP3-first	2	14.82	77.90%	32.52%
D.106	CNN	1	6.09	91.20%	-
D.107	CNN	2	16.59	91.60%	18.35%

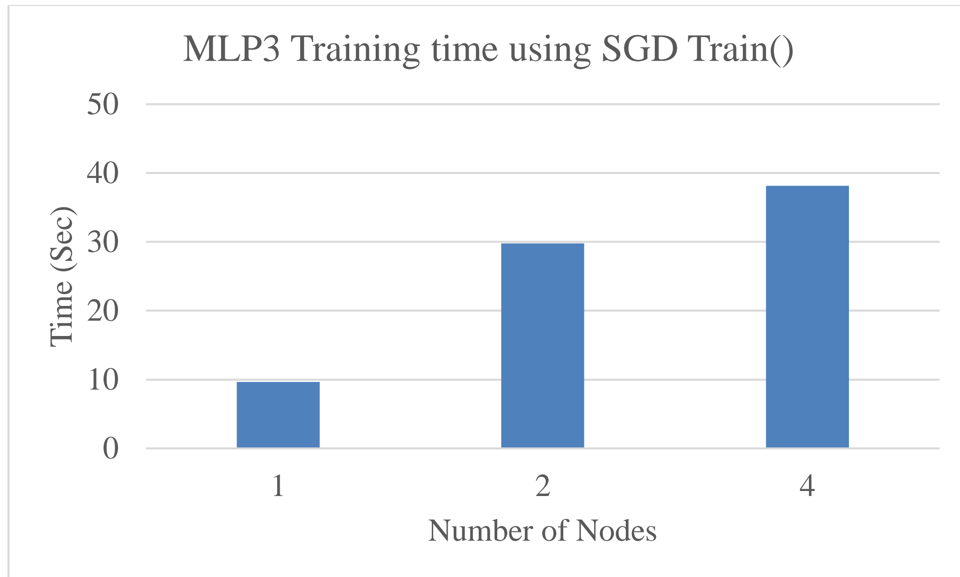

Figure 4.2. Training time using Auto SGD Train() with MLP3 and 4 nodes. Demonstrates training time increases as nodes are added due to communication overhead.

4.2.7 *Model Parallel Unique Attribute (Model Size)*

A. **Main Topic:** GPU parallelization allows models too big for a single node to be trained.

B. **Supporting Data:** Table 4.23 illustrates two important facts. First, as model complexity increases GPU parallelization efficiency increases with it. Second, the limited resources environment of the experiments restricted the size of the model trained and prohibited larger sizes to be tested.

C. **Why this happens:** GPU memory restricts larger models from training on a single node. Without this comparison, the speedup efficiency is unable to be attained. It is estimated, further tests would show an efficiency greater than 50%, if the current trend continued. This failure also demonstrates an important benefit of the parallelization. Models which require more resources than a single node can provide are able to be trained when model parallelization is used. Different from data parallel, model parallelism reduces the requirements on a single node. By splitting the neurons across multiple nodes, each node is only required to store and compute a subsection of the total network. If a network required 4 GB to store and each TX1 GPU can only store 2GB, then a minimum of 2 nodes is required to store and train the model.

   Although MLP3 is a poor example of when such complexity would be needed, complex models outside of our experiments could benefit from this. ResNet [47] and VGG16 [2] have incredibly large fully-connected classifier layers. VGG16 for instance uses a three-layer classifier of 25088->4096->4096. This model is too large to be loaded on a single TX1 node but can be loaded and trained across four nodes.

D. **Drawbacks/Limitations:** Speedup efficiency is still less than optimal.

E. **Conclusion/Summary:** Although GPU parallelization efficiency is less than optimal in our tests, it does allow large, complex models to be trained where resources requirements are greater than a single node can provide.

Table 4.23. MLP3 GPU model complexity SGD Train() compared with speedup efficiency trained on MNIST dataset.

MLP3 Model Complexity Comparison on GPU with SGD Train()						
Appx. Ref.	**Nodes**	**SLS**	**TLS**	**Time (sec)**	**Accuracy**	**Efficiency**
D.84	1	2048	2048	13.82	74.80%	-
D.85	2	2048	2048	29.38	71.10%	23.52%
D.86	1	4096	2048	26.63	70.80%	-
D.87	2	4096	2048	43.95	69.80%	30.30%
D.88	1	4096	4096	42.86	57.20%	-
D.89	2	4096	4096	51.53	68.70%	41.59%
D.90	1	8192	4096	Error: killed		
D.91	2	8192	4096	68.92	72.40%	#VALUE!
D.92	1	8192	8192	Error: Not enough RAM		
D.93	2	8192	8192	109.2	58.20%	#VALUE!

Chapter 5. CONCLUSION

For the conclusion, we start by summarizing the takeaways and discussing the value added from our research. Next, the personal effect of the research on the author is covered. Followed by a bulleted outline of limitations and future work.

5.1 RESULT STATEMENT

This work added value in accurately measuring software complexity and the importance of reducing not only execution complexity but also increasing usability to the user. Halstead's complexity measures provide an excellent, uniform method to measure usability without bias and consistently. As layers continue to be built upon other layers of abstraction the user becomes further removed from the underlying components. This allows a machine learning engineer to train neural networks without worrying how to establish a TCP connection or address a specific memory location. Continuing to build these layers allows for further technological advances in every field from biology to farming and critical to our development as a society and as a species.

TorchAD-NN data parallel results demonstrate implementation complexity (Usability) is greatly reduced when compared to manual parallelization and slightly reduced when compared to GPU parallelization. Data parallelism achieves similar efficiency when the same training is conducted manually and offers an increase in efficiency for pre-processing. Our TorchAD-NN data parallel module successfully achieved our research criteria by increasing usability and maintaining speedup efficiency of distributed neural network training.

TorchAD-NN model parallel also demonstrates a reduction in implementation complexity. These results are a bit more difficult to interpret as the code required is the same, only the perspective from the user is changed. TorchAD-NN model parallelism maintained the

same efficiency when compared to manual parallelization, yet overall was less efficient than single training. The benefit is larger models can be trained which would otherwise be impossible on a single node. TorchAD-NN model parallel module successfully reduced complexity and maintained efficiency when compared with manual parallelization.

The model parallel approach brings incredible flexibility and power to the fingertips of machine learning engineers. These engineers can now create networks split across multiple nodes with ease and exact control on which layers are split and synchronized. CIFAR and MNIST did not fully exemplify the power of this module. Despite the convolution layer limitation, many networks such as ResNet [47], VGG16 [2], Deep Speech 2 [55], and more can be trained where a single node would be unable. As discussed further under Section 5.5, the restriction is the overall efficiency of model parallelization (not unique to TorchAD-NN) and we desire to continue work on this module and support additional component's parallelization.

In summary, TorchAD-NN automation greatly reduced the implementation complexity of distributing neural network training across multiple devices without loss of computational efficiency when compared to manual parallelization. TorchAD-NN is released to the public and supports both data and model parallelism in Torch 7.0. As a species, we continue our pursuit for superintelligence and the master algorithm. Reducing complexity and support for large models and datasets is another step in this notable and challenging pursuit.

5.2 Stakeholder Effect

TorchAD-NN targets a specific group of engineers and scientists. This work is aimed at helping those in small departments with low funding, scientists working in fields unrelated to computer science, and those working in machine learning as a hobbyist who lack the funding and access to

large-scale GPU clusters. TorchAD-NN also targets machine learning and data scientists who do not have a strong foundation in software engineer and distributed computing. For those mentioned, this library brings the ability to distribute neural network training across a small-scale cluster of GPUs and requires zero distributed systems programming knowledge. This distribution decreases the training time required and allows larger, more complex models to be trained, opening more avenues of research and innovation.

5.3 Personal Effect

This project helped me grow immensely as a researcher, machine learning scientist, and software engineer. I approached this project of automatic parallelization with a strictly software engineering mindset of how to solve the challenges with distributed neural network training (where many others use a mathematical approach). I started with nothing and built from the ground up the hardware setup and mastered each of the many layers necessary to build TorchAD-NN. I learned the importance of doing research then constructing my hypothesis, documenting work as I go, pre-planning experiments, conducting all experiments before revision, stepping back to analyze results, and only then possibly setup new experiments.

Here at the end, I feel I have reached the verge of ground breaking research yet was left without time to climb over. I couldn’t have done it differently, it took the more than 1 years’ worth of work to get to the point where I understand the field enough to feel I can truly contribute. Next, I move forward into my new job and career with a solid foundation in machine learning, software engineering, and scientific research. These skills are invaluable for my future and well worth the effort to attain them.

5.4 LIMITATIONS

Limitations on our research are discussed in the following bullets.

- The CNNs and MLPs used are of limited complexity and did not fully represent the capabilities of TorchAD-NN. Further experiments using models, such as ResNet, VGG16, and Deep Speech 2, would better demonstrate its capabilities.
- Small datasets (less than 100K) led to a notable accuracy drop when using the data parallel approach. Most of the model accuracy is achieved with the small data sample and synchronization shows little improvement. Using a smaller dataset on each node already led to an impacted model accuracy and thus synchronization showed little improvement. Larger datasets would reduce this impact by leaving a larger chunk on each machine.
- Because we used a custom setup, our experiments were limited to execution on the Jetson TX1 and with a maximum of 4 nodes. More nodes and different types of hardware would yield more confidence global applicability.
- Convolution layers were unable to be parallelized in TorchAD-NN. This limits the types and methods of model parallelization. We were restricted to only fully-connected layers. Parallelization of more components of the CNN would give increased flexibility and applicability to TorchAD-NN.
- OpenMPI is required for underlying communication in TorchMPI and TorchAD-NN. OpenMPI can be difficult to install, configure, and requires duplicates of code and data be present on each device. The ability to provide communication without OpenMPI and automatically handle data and code distribution would be an added gain to the "Automaticity" of TorchAD-NN.

5.5 FUTURE WORK AND NEXT STEPS

Future work of our research is discussed in the following bullets.

- Combine both approaches and test usability and efficiency of such an approach.
- Test CIFAR dataset and CNN networks with TorchAD-NN model parallel.
- Test ResNet, VGG16, and Deep Speech 2 (and associated datasets, e.g., ImageNet) with TorchAD-NN model parallel.
- Override existing C and CUDA objects present for THNN to allow spatial convolution parallelism.
- Investigate using data parallel with other machine learning techniques outside of neural networks.
- Auto tune synchronization frequency as neural network model is being trained.
- Use other popular DNNs from Github-soumith/imagenet-multigpus to further test TorchAD-NN
- Investigate TorchMPI hanging on GPU memory initialization and attempt to fix the bug.
- Perform power consumption analysis on Cobra Cluster with different parallelization techniques.
- Prototype and investigate data shuffle split technique (where data is spilt in a shuffled manner rather than in complete chunks when performing data parallelization).

BIBLIOGRAPHY

[1] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015.

[2] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” *ArXiv14091556 Cs*, Sep. 2014.

[3] J. Dean *et al.*, “Large Scale Distributed Deep Networks,” in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1223–1231.

[4] J. Dean, “Large Scale Deep Learning,” presented at the CIKM 2014 conference, Tsinghua University, 2014.

[5] M. Abadi *et al.*, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems,” *ArXiv160304467 Cs*, Mar. 2016.

[6] T. Little, “Context-adaptive agility: managing complexity and uncertainty,” *IEEE Softw.*, vol. 22, no. 3, pp. 28–35, May 2005.

[7] R. D. Banker, G. B. Davis, and S. A. Slaughter, “Software Development Practices, Software Complexity, and Software Maintenance Performance: A Field Study,” *Manag. Sci.*, vol. 44, no. 4, pp. 433–450, Apr. 1998.

[8] V. Antinyan *et al.*, “Identifying complex functions: By investigating various aspects of code complexity,” in *2015 Science and Information Conference (SAI)*, 2015, pp. 879–888.

[9] J. Keuper and F. J. Preundt, “Distributed Training of Deep Neural Networks: Theoretical and Practical Limits of Parallel Scalability,” in *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, 2016, pp. 19–26.

[10] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng, “On Optimization Methods for Deep Learning,” in *Proceedings of the 28th International Conference on International Conference on Machine Learning*, USA, 2011, pp. 265–272.

[11] C. Szegedy *et al.*, “Going Deeper with Convolutions,” *ArXiv14094842 Cs*, Sep. 2014.

[12] J. Dean, *CIKM ’14: Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. New York, NY, USA: ACM, 2014.

[13] R. Gu, F. Shen, and Y. Huang, “A parallel computing platform for training large scale neural networks,” in *2013 IEEE International Conference on Big Data*, 2013, pp. 376–384.

[14] S. Li, J. He, Y. Li, and M. U. Rafique, “Distributed Recurrent Neural Networks for Cooperative Control of Manipulators: A Game-Theoretic Perspective,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 28, no. 2, pp. 415–426, Feb. 2017.

[15] S. Teerapittayanon, B. McDanel, and H. T. Kung, “Distributed Deep Neural Networks Over the Cloud, the Edge and End Devices,” in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 328–339.

[16] F. N. Iandola, K. Ashraf, M. W. Moskewicz, and K. Keutzer, “FireCaffe: near-linear acceleration of deep neural network training on compute clusters,” *ArXiv151100175 Cs*, Oct. 2015.

[17] A. Abdelfattah, “Image Classification using Deep Neural Networks — A beginner friendly approach using TensorFlow,” *Medium*, 27-Jul-2017. .

[18] R. Collobert, S. Bengio, and J. Marithoz, *Torch: A Modular Machine Learning Software Library*. 2002.

[19] “TorchMPI: Implements a message passing interface (MPI),” 14-Nov-2017. [Online]. Available: https://github.com/facebookresearch/TorchMPI.

[20] G. K. Gill and C. F. Kemerer, "Cyclomatic complexity density and software maintenance productivity," *IEEE Trans. Softw. Eng.*, vol. 17, no. 12, pp. 1284–1288, Dec. 1991.

[21] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[22] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," *Univ. Tor.*, May 2012.

[23] N. Grabaskas and D. Si, "Anomaly Detection from Kepler Satellite Time-Series Data," in *Machine Learning and Data Mining in Pattern Recognition*, 2017, pp. 220–232.

[24] L. Paul, "Current Trends in IT Notes on Neural Nets." [Online]. Available: http://users.ecs.soton.ac.uk/phl/ctit/nn/nn.html. [Accessed: 19-Jan-2018].

[25] I. Aleksander and H. Morton, *An Introduction to Neural Computing*. New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[26] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Netw.*, vol. 61, pp. 85–117, Jan. 2015.

[27] N. Wolchover, "New Theory Cracks Open the Black Box of Deep Neural Networks | WIRED." [Online]. Available: https://www.wired.com/story/new-theory-deep-learning/. [Accessed: 19-Jan-2018].

[28] L. Reading-Ikkanda, *Quanta Magazine*. 2017.

[29] "Unsupervised Feature Learning and Deep Learning Tutorial." [Online]. Available: http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/. [Accessed: 19-Jan-2018].

[30] A. Coates and A. Y. Ng, "Multi-camera object detection for robotics," in *2010 IEEE International Conference on Robotics and Automation*, 2010, pp. 412–419.

[31] B. Kim, Y. Jeon, H. Park, D. Han, and Y. Baek, "Design and Implementation of the Vehicular Camera System Using Deep Neural Network Compression," in *Proceedings of the 1st International Workshop on Deep Learning for Mobile Systems and Applications*, New York, NY, USA, 2017, pp. 25–30.

[32] M. D. Zeiler and R. Fergus, "Visualizing and Understanding Convolutional Networks," *ArXiv13112901 Cs*, Nov. 2013.

[33] S. Tripathi, G. Dane, B. Kang, V. Bhaskaran, and T. Nguyen, "LCDet: Low-Complexity Fully-Convolutional Neural Networks for Object Detection in Embedded Systems," in *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2017, pp. 411–420.

[34] M. Lin, Q. Chen, and S. Yan, "Network In Network," *ArXiv13124400 Cs*, Dec. 2013.

[35] J. Deng, W. Dong, R. Socher, L. J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[36] A. Sergeev, "Meet Horovod: Uber's Open Source Distributed Deep Learning Framework for TensorFlow," *Uber Engineering Blog*, 17-Oct-2017. [Online]. Available: https://eng.uber.com/horovod/. [Accessed: 22-Jan-2018].

[37] "Torch | Scientific computing for LuaJIT." [Online]. Available: http://torch.ch/. [Accessed: 29-Nov-2017].

[38] "TensorFlow," *TensorFlow*. [Online]. Available: https://www.tensorflow.org/. [Accessed: 29-Nov-2017].

[39] "Lua vs Python 3 (64-bit Ubuntu quad core) | Computer Language Benchmarks Game." [Online]. Available:

http://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=lua&lang2=python3. [Accessed: 29-Nov-2017].

[40] "lua-users wiki: Lua Versus Python." [Online]. Available: http://lua-users.org/wiki/LuaVersusPython. [Accessed: 29-Nov-2017].

[41] "Slant - Python vs Lua detailed comparison as of 2017," *Slant*. [Online]. Available: https://www.slant.co/versus/110/1418/~python_vs_lua. [Accessed: 29-Nov-2017].

[42] "Is TensorFlow better than other leading libraries such as Torch/Theano?" [Online]. Available: https://www.quora.com/Is-TensorFlow-better-than-other-leading-libraries-such-as-Torch-Theano. [Accessed: 29-Nov-2017].

[43] A. Torralba, R. Fergus, and W. T. Freeman, "80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, no. 11, pp. 1958–1970, Nov. 2008.

[44] G. A. Miller, "WordNet: A Lexical Database for English," *Commun ACM*, vol. 38, no. 11, pp. 39–41, Nov. 1995.

[45] A. C. Berg, T. L. Berg, and J. Malik, "Shape matching and object recognition using low distortion correspondences," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, 2005, vol. 1, pp. 26–33 vol. 1.

[46] J. Chen, Y. Wang, Y. Wu, and C. Cai, "An Ensemble of Convolutional Neural Networks for Image Classification Based on LSTM," in *2017 International Conference on Green Informatics (ICGI)*, 2017, pp. 217–222.

[47] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[48] L. Bottou, "Large-Scale Machine Learning with Stochastic Gradient Descent," in *Proceedings of COMPSTAT'2010*, Physica-Verlag HD, 2010, pp. 177–186.

[49] "TorchMPI: Model Parallel Example," 16-Jan-2018. [Online]. Available: https://github.com/facebookresearch/TorchMPI.

[50] S. Zhang, A. Choromanska, and Y. LeCun, "Deep learning with Elastic Averaging SGD," *ArXiv14126651 Cs Stat*, Dec. 2014.

[51] T. J. McCabe, "A Complexity Measure," *IEEE Trans Softw Eng*, vol. 2, no. 4, pp. 308–320, Jul. 1976.

[52] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977.

[53] "IBM Knowledge Center - Halstead Metrics." [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSSHUF_8.0.0/com.ibm.rational.testrt.studio.doc/topics/csmhalstead.htm. [Accessed: 01-Dec-2017].

[54] "SpatialConvolutionMM.lua Source Code," 28-Jan-2018. [Online]. Available: https://github.com/torch/nn. [Accessed: 01-Feb-2018].

[55] D. Amodei *et al.*, "Deep Speech 2: End-to-End Speech Recognition in English and Mandarin," *ArXiv151202595 Cs*, Dec. 2015.

APPENDIX A – SOURCE CODE

Data Parallel Module Source Code

```
local datamodule = {}


local dataShuffle = false  --shuffle data when splitting
------------------------------------------------------------------
-- Name:       parallelize
-- Inputs:     data array, targets array, ANN model, data array size, mpi, mpinn, batchSize
-- Outputs: new data array, new targest array, optimized batch size, new data size
-- Summary: This function receives elements from the user and calls the necessary
--                    functions to prepare data parallelization across the MPI nodes.
--
------------------------------------------------------------------


function datamodule.parallelize( data, targets, model, size, mpi_obj, mpinn_obj, batchSize
)

       -- check required parameters were passed
       if (data == nil or targets == nil or model == nil or size == nil) then
              print ("-usage for parallelize(data, targets, model, size)")
              return -1
       end

       -- if no MPI object is passed create one
       if (mpi_obj == nil) then
              require 'torchmpi'
              mpi_obj = require('torchmpi')
              mpi_obj.start(true)  --true equals use GPU
       end

       -- if no MPI NN object is passed create one
       if (mpinn_obj == nil) then
              mpinn_obj = require('torchmpi.nn')
              mpinn_obj.synchronizeParameters(model)
       end

       -- split data and targets across all nodes
       local newdata, dataSize = datamodule.data_parallel(data, size, mpi_obj)
       local newtargets = datamodule.data_parallel(targets, size, mpi_obj)
```

```
        -- determine speed
        -- this option removed from current implementation
        -- local speed = datamodule.comm_speed(data, targets, model, mpi_obj, mpinn_obj)

        -- determine optimal batch size
        datamodule.optimize_sync(speed, dataSize, model, mpinn_obj, mpi_obj, batchSize)

        -- ensure all ranks are complete before returning
        mpi_obj.barrier()

        return newdata, newtargets, dataSize
end




------------------------------------------------------------------
-- Name:       optimize_sync
-- Inputs:     communication speed, data array size
-- Outputs: optimized batch size
-- Summary: This functions returns the optimized batch size based
--                      on communication speed and data array size.
--
------------------------------------------------------------------


function datamodule.optimize_sync( speed, size, model, mpinn, mpi, batchSize )

        -- if batchSize is not given it will be optimized
        if (batchSize == nil) then
                if (size < 1000) then
                        batchSize = 1
                elseif (size < 2500) then
                        batchSize = 10
                elseif (size < 5000) then
                        batchSize = 50
                else
                        batchSize = 100
                end
        end

        --
```

```
-- After comm test and batchSize determined override backward propogation function
to syncGradients
--
function nn.Sequential:backward(input, gradOutput, scale)
        scale = scale or 1
        local currentGradOutput = gradOutput
        local currentModule = self.modules[#self.modules]
        for i=#self.modules-1,1,-1 do
                local previousModule = self.modules[i]
                currentGradOutput = self:rethrowErrors(currentModule, i+1,
                    'backward', previousModule.output, currentGradOutput, scale)
                currentModule.gradInput = currentGradOutput
                currentModule = previousModule
        end
        currentGradOutput = self:rethrowErrors(currentModule, 1, 'backward', input,
                currentGradOutput, scale)
        self.gradInput = currentGradOutput

        -- additional sync functionality added
        -- *
        if (self.sync_counter == nil) then
                self.sync_counter = 1;
        end

        -- sync and shutdown when dataset is complete
        if (self.sync_counter == size) then
                mpinn.synchronizeGradients(model)
                mpi.stop()
        elseif (self.sync_counter % batchSize == 0) then -- sync at end of batch
                mpinn.synchronizeGradients(model)
        end
        self.sync_counter = self.sync_counter + 1
        -- *
        -- end of additional functionality

        return currentGradOutput
end

--This function needs to be overriden to allow for stochastic gradient training
function nn.StochasticGradient:train(dataset)
   local iteration = 1
   local currentLearningRate = self.learningRate
   local module = self.module
```

```
local criterion = self.criterion


local shuffledIndices = torch.randperm(dataset:size(), 'torch.LongTensor')
if not self.shuffleIndices then
      for t = 1,dataset:size() do
            shuffledIndices[t] = t
      end
end


print("# StochasticGradient: training")


while true do
      local currentError = 0
      for t = 1,dataset:size() do
            local example = dataset[shuffledIndices[t] ]
            local input = example[1]
            local target = example[2]


            currentError = currentError +
                   criterion:forward(module:forward(input), target)


            module:updateGradInput(input,
                   criterion:updateGradInput(module.output, target))
            module:accUpdateGradParameters(input, criterion.gradInput,
                   currentLearningRate)

           -- additional sync functionality added
           -- *
           if (self.sync_counter == nil) then
                  self.sync_counter = 1
           end

           -- sync and shutdown when dataset is complete
           if (self.sync_counter == size) then
                  mpinn.synchronizeGradients(model)
           elseif (self.sync_counter % batchSize == 0) then --sync at end batch

                  mpinn.synchronizeGradients(model)
```

```
                end
                self.sync_counter = self.sync_counter + 1
                -- *
                -- end of additional functionality


                 if self.hookExample then
                       self.hookExample(self, example)
                 end
          end


          currentError = currentError / dataset:size()


          if self.hookIteration then
                self.hookIteration(self, iteration, currentError)
          end


          if self.verbose then
                print("# current error = " .. currentError)
          end
          iteration = iteration + 1
          currentLearningRate =
                self.learningRate/(1+iteration*self.learningRateDecay)
          if self.maxIteration > 0 and iteration > self.maxIteration then
                print("# StochasticGradient: you have reached the maximum number of
                    iterations")
                print("# training error = " .. currentError)
                break
          end
    end
 end


 return batchSize
end



----------------------------------------------------------------------------------------
```

```
-- Name:       data_parallel
-- Inputs:     array, array size
-- Outputs: new array, new size
-- Summary: This function splits the data array evenly across the MPI nodes.
--                    Any remainder is given to the last node.
--
-----------------------------------------------------------------


function datamodule.data_parallel( data, size, mpi )

        -- determine which rank will get which data
        -- copy data from dataset to newdataset from start to end
        local remainder = 0
        -- how many elements will be placed on each rank, remainder elements go to last rank
        local stripe = ( size - ( size % mpi.size() ) ) / mpi.size()
        size = size - remainder
        -- where will this rank's data start at
        local start  = ( mpi.rank() * stripe ) + 1
        -- where will this rank's data end at
        local finish = start + (stripe - 1) + remainder

        -- create local var for new dataset
        local newdata = {}

        newdata = data[{ {start,finish} } ]

        if dataShuffle then
                local index = 1
                for i = 1,size,mpi.size() do
                        newdata[ { { index,index } } ] = data[ { { i + mpi.rank(), i +
                                mpi.rank() } } ]
                        index = index + 1
                end
        end

        print ("Rank: " .. mpi.rank() .. " Start Point: " .. start .. " End Point: " ..
                finish .. " Stripe: " .. stripe .. " Remainder: " .. remainder)

        -- return new dataset size
        local dataSize = stripe + remainder

        return newdata, dataSize
```

```
end



-----------------------------------------------------------------
-- Name:       comm_speed
-- Inputs:     data array, target array, ANN model
-- Outputs: communication speed
-- Summary:    This function runs ten test of the network forward and backward
--                   propogation with a sync of gradients across nodes. This is done
--                   10 times and the average time returned.
--
-----------------------------------------------------------------


function datamodule.comm_speed( data, targets, model, mpi, mpinn )


       -- ensure all ranks are ready before beginning comm check
       mpi.barrier()

       local timer = torch.Timer()
       timer:stop()
       timer:resume()

       for i = 1,10 do
              local output = model:forward(data[i])
              local df_do  = criterion:backward(output, targets[i])
              model:backward(data[i], df_do)
              mpinn.synchronizeGradients(model)
       end

       timer:stop()

       local speed = (timer:time().real) / 10

       print("Rank: " .. mpi.rank() .. " comm speed: " .. speed)

       return speed
end


return datamodule
```

Model Parallel Module Source Code

```lua
local nodemodule = {}


local printDims   = false  -- For debugging information
local syncTanh    = false  -- Experimental tanh all gather
local syncReshape = false  -- Experimental reshape all gather


------------------------------------------------------------------
-- Name:       Narrow Input
-- Inputs:     Input array
-- Outputs: Narrowed input array
-- Summary: This function accepts the normal input array and returns the
--                  the reduced array that is specific to each node.
--
------------------------------------------------------------------


local function narrowInput(input)
   local dim = input:nDimension()
   assert(input:size(dim) % mpi.size() == 0)
   local size = input:size(dim) / mpi.size()
   return input:narrow(dim, mpi.rank() * size + 1, size)
end


------------------------------------------------------------------
-- Name:       Initial Linear Layer
-- Inputs:     No additional input is required to use this layer
-- Outputs: Standard output from nn.Linear is used
-- Summary: These functions override the original and allow the input layer to
--                  be split over multiple nodes within the network. Output is then
--                  Synchronized using allreduceTensor(). For the initial linear layer
--                  accGradParameters() input must also be narrowed.
--
------------------------------------------------------------------


local MPInitialLinear, parent = torch.class('nn.MPInitialLinear', 'nn.Linear')
```

```
function MPInitialLinear.__init(self, i, o)
   assert(i % mpi.size() == 0, ('i=%d not divisible by %d'):format(i, mpi.size()))
   nn.Linear.__init(self, i / mpi.size(), o)
end


function MPInitialLinear.updateOutput(self, input)
        local input = narrowInput(input)
        if printDims then
                print('#INITIAL LINEAR UPDATE OUTPUT#')
                print('*****input*****')
                print(input:size())
        end
        self.output = nn.Linear.updateOutput(self, input)
        if printDims then
                print('*****output*****')
                print(self.output:size())
        end
        mpi.allreduceTensor(self.output)
        return self.output
end


function MPInitialLinear.updateGradInput(self, input, gradOutput)
        local input = narrowInput(input)
        if printDims then
                print('#INITIAL LINEAR UPDATE GRAD INPUT#')
                print('*****input*****')
                print(input:size())
                print('*****grad output*****')
                print(gradOutput:size())
        end
        self.gradInput = nn.Linear.updateGradInput(self, input, gradOutput)
        mpi.allreduceTensor(self.gradInput)
        return self.gradInput
end


function MPInitialLinear.accGradParameters(self, input, gradOutput, scale)
   local input = narrowInput(input)
   nn.Linear.accGradParameters(self, input, gradOutput, scale)
end
```

```
---------------------------------------------------------------
-- Name:       Base Linear Layer
-- Inputs:     No additional input is required to use this layer
-- Outputs: Standard output from nn.Linear is used
-- Summary: These functions override the original and allow the input layer to
--                    be split over multiple nodes within the network. Output is then
--                    Synchronized using allreduceTensor(). For base linear layer,
--                    accGradParameters() input is not narrowed.
--
---------------------------------------------------------------


local MPBaseLinear, parent = torch.class('nn.MPBaseLinear', 'nn.Linear')
function MPBaseLinear.__init(self, i, o)
   assert(i % mpi.size() == 0, ('i=%d not divisible by %d'):format(i, mpi.size()))
   dimension_used = i / mpi.size()
   nn.Linear.__init(self, dimension_used, o)
end


function MPBaseLinear.updateOutput(self, input)
   local input = narrowInput(input)
   self.output = nn.Linear.updateOutput(self, input)
   if printDims then
              print('#BASE LINEAR UPDATE OUTPUT#')
              print('*****input*****')
              print(input:size())
              print('*****output*****')
              print(self.output:size())
       end
       mpi.allreduceTensor(self.output)
       return self.output
end


function MPBaseLinear.updateGradInput(self, input, gradOutput)
       local input = narrowInput(input)
       if printDims then
              print('#BASE LINEAR UPDATE GRAD INPUT#')
              print('*****input*****')
              print(input:size())
              print('*****grad output*****')
```

```
                print(gradOutput:size())
        end
        self.gradInput = nn.Linear.updateGradInput(self, input, gradOutput)
        mpi.allreduceTensor(self.gradInput)
        return self.gradInput
end


------------------------------------------------------------------
-- Name:       Tanh Layer
-- Inputs:     No additional input is required to use this layer
-- Outputs: Standard output from nn.Tanh is used
-- Summary: These functions narrow the self.output from updateGradInput()
--                    during backward propagation. Output from updateGradInput() is
--                    expanded for the above linear layer during backward propagation.
--                    An experiment allgather during this expansion can be turned on.
--
------------------------------------------------------------------


local MPTanh, parent = torch.class('nn.MPTanh', 'nn.Tanh')
function MPTanh:updateOutput(input)
        self.output = nn.Tanh.updateOutput(self, input)
        if printDims then
                print('#TANH UPDATE OUTPUT#')
                print('*****input*****')
                print(input:size())
                print('*****output*****')
                print(self.output:size())
        end
        return self.output
end


function MPTanh:updateGradInput(input, gradOutput)
        self.output = narrowInput(self.output)
        if printDims then
                print('#TANH UPDATE GRAD INPUT#')
                print('*****input*****')
                print(input:size())
                print('*****output*****')
                print(self.output:size())
        end
```

```
        input.THNN.Tanh_updateGradInput(
          input:cdata(),
          gradOutput:cdata(),
          self.gradInput:cdata(),
          self.output:cdata()
        )
        -- need to get gradInput from each node and concatenate ######
        tempGradInput = self.gradInput
        for i=2,mpi.size() do
                tempGradInput = torch.cat(tempGradInput, self.gradInput)
        end
        if mpi.size() >  1 and syncTanh then
                mpi.allgatherTensor(self.gradInput, tempGradInput)
        end
        self.gradInput = tempGradInput
        if printDims then
                print('*****grad input*****')
                print(self.gradInput:size())
        end
        return self.gradInput
end


-----------------------------------------------------------------
-- Name:       Initial Reshape Layer
-- Inputs:     No additional input is required to use this layer
-- Outputs: Standard output from nn.Reshape is used
-- Summary: The below functions narrows the input when Reshape.updateGradInput is
--                    used at the top of the network. This corresponds to the
--                    narrowed out from the below linear layer's updateGradInput().
--
-----------------------------------------------------------------


local MPInitialReshape, parent = torch.class('nn.MPInitialReshape', 'nn.Reshape')


function MPInitialReshape:updateOutput(input)
   if not input:isContiguous() then
      self._input = self._input or input.new()
      self._input:resizeAs(input)
      self._input:copy(input)
      input = self._input
```

```
   end


   if (self.batchMode == false) or (
         (self.batchMode == nil) and
         (input:nElement() == self.nelement and input:size(1) ~= 1)
      ) then
      self.output:view(input, self.size)
   else
      self.batchsize[1] = input:size(1)
      self.output:view(input, self.batchsize)
   end
   return self.output
end


function MPInitialReshape:updateGradInput(input, gradOutput)
       local input = narrowInput(input)
       if printDims then
              print('#INITIAL RESHAPE UPDATE GRAD INPUT#')
              print('*****input*****')
              print(input:size())
              print('*****output*****')
              print(gradOutput:size())
       end
   if not gradOutput:isContiguous() then
      self._gradOutput = self._gradOutput or gradOutput.new()
      self._gradOutput:resizeAs(gradOutput)
      self._gradOutput:copy(gradOutput)
      gradOutput = self._gradOutput
   end


   self.gradInput:viewAs(gradOutput, input)
   return self.gradInput
end


-------------------------------------------------------------------
-- Name:       Base Reshape Layer
-- Inputs:     No additional input is required to use this layer
-- Outputs: Standard output from nn.Reshape is used
-- Summary: Output from updateGradInput() is expanded for the above linear
```

```
--                    layer during backward propagation. An experiment allgather
--                    during this expansion can be turned on.
--
-------------------------------------------------------------------


local MPBaseReshape, parent = torch.class('nn.MPBaseReshape', 'nn.Reshape')


function MPBaseReshape:updateOutput(input)
   if not input:isContiguous() then
      self._input = self._input or input.new()
      self._input:resizeAs(input)
      self._input:copy(input)
      input = self._input
   end


   if (self.batchMode == false) or (
         (self.batchMode == nil) and
         (input:nElement() == self.nelement and input:size(1) ~= 1)
      ) then
      self.output:view(input, self.size)
   else
      self.batchsize[1] = input:size(1)
      self.output:view(input, self.batchsize)
   end
   return self.output
end


function MPBaseReshape:updateGradInput(input, gradOutput)
        tempGrad = gradOutput
        for i=2,mpi.size() do
                tempGrad = torch.cat(tempGrad, gradOutput)
        end
        if mpi.size() >  1 and syncReshape then
                mpi.allgatherTensor(gradOutput, tempGrad)
        end
        gradOutput = tempGrad
        if printDims then
                print('#BASE RESHAPE UPDATE GRAD INPUT#')
                print('*****input*****')
```

```
            print(input:size())
            print('*****output*****')
            print(gradOutput:size())
        end
    if not gradOutput:isContiguous() then
        self._gradOutput = self._gradOutput or gradOutput.new()
        self._gradOutput:resizeAs(gradOutput)
        self._gradOutput:copy(gradOutput)
        gradOutput = self._gradOutput
    end


    self.gradInput:viewAs(gradOutput, input)
    return self.gradInput
end


return nodemodule
```

APPENDIX B – NEURAL NETWORK TRAINING SETUP

CIFAR10 – Canadian Institute for Advanced Research 10

Data – from “http://torch7.s3-website-us-east-1.amazonaws.com/data/cifar-10-torch.tar.gz”

CNN Model: adapted from https://github.com/spdjudd/Torch-Classifier/blob/master/model.lua

```
nn.Sequential {
[input -> (1) -> (2) -> (3) -> (4) -> (5) -> (6) -> (7) -> (8) -> (9) -> (10) -> (11) -> output]
    (1): nn.SpatialConvolutionMM(3 -> 64, 5x5)
    (2): nn.ReLU
    (3): nn.SpatialMaxPooling(3x3, 3,3)
    (4): nn.SpatialConvolutionMM(64 -> 64, 5x5)
    (5): nn.ReLU
    (6): nn.SpatialMaxPooling(3x3, 3,3)
    (7): nn.View(64)
    (8): nn.Dropout(0.500000)
    (9): nn.Linear(64 -> 100)
    (10): nn.ReLU
    (11): nn.Linear(100 -> 10)
}
```

MNIST - Modified National Institute of Standards and Technology database

Data – from https://s3.amazonaws.com/torch7/data/mnist.t7.tgz

CNN Model- adapted from https://github.com/torch/demos/blob/master/train-a-digit-classifier/train-on-mnist.lua

```
nn.Sequential {
[input -> (1) -> (2) -> (3) -> (4) -> (5) -> (6) -> (7) -> (8) -> (9) -> (10) -> output]
    (1): nn.SpatialConvolutionMM(1 -> 32, 5x5)
    (2): nn.Tanh
    (3): nn.SpatialMaxPooling(3x3, 3,3, 1,1)
    (4): nn.SpatialConvolutionMM(32 -> 64, 5x5)
    (5): nn.Tanh
    (6): nn.SpatialMaxPooling(2x2, 2,2)
```

(7): nn.Reshape(576)

(8): nn.Linear(576 -> 200)

(9): nn.Tanh

(10): nn.Linear(200 -> 10)

}

MLP2 Model- adapted from https://github.com/torch/demos/blob/master/train-a-digit-classifier/train-on-mnist.lua

```
nn.Sequential {
[input -> (1) -> (2) -> (3) -> (4) -> output]
    (1): nn.MPInitialReshape(1024)
    (2): nn.MPInitialLinear(1024 -> 2048)
    (3): nn.MPTanh
    (4): nn.MPBaseLinear(2048 -> 10)
}
```

MLP3 Model- adapted from https://github.com/torch/demos/blob/master/train-a-digit-classifier/train-on-mnist.lua

```
nn.Sequential {
[input -> (1) -> (2) -> (3) -> (4) -> (5) -> (6) -> output]
    (1): nn.MPInitialReshape(1024)
    (2): nn.MPInitialLinear(1024 -> 2048)
    (3): nn.MPTanh
    (4): nn.MPBaseLinear(2048 -> 1024)
    (5): nn.MPTanh
    (6): nn.MPBaseLinear(1024 -> 10)
}
```

APPENDIX C – TEST RESULTS (DATA PARALLEL)

Ref	CIFAR-10								
	Nodes	Batch Size	Pre-pro (sec)	Train (sec)	Accuracy	SU-P	SU-T	Acc-Loss	Efficiency
	GPU SGD Train()								
1	1	100	68.81	547.18	48.57%	-	-	-	-
2	1	500	67.61	545.71	47.00%	-	-	-	-
3	1	1,000	68.24	533.00	47.71%	-	-	-	-
4	1	500	48.05	264.51	39.22%	-	-	-	-
5	1	500	38.05	136.78	34.06%	-	-	-	-
	Auto SGD Train()								
6	2	100	48.96	559.81	39.98%	1.41	0.98	-8.59%	48.87%
7	2	500	49.03	333.97	41.40%	1.38	1.63	-5.60%	81.70%
8	2	1,000	49.25	326.16	39.51%	1.39	1.63	-8.20%	81.71%
9	4	100	38.93	441.06	33.68%	1.77	1.24	-14.89%	31.02%
10	4	500	39.22	202.89	33.32%	1.72	2.69	-13.68%	67.24%
11	4	1,000	39.23	177.26	34.28%	1.74	3.01	-13.43%	75.17%
	Manual SGD Train()								
12	2	100	49.09	579.44	40.60%	1.40	0.94	-7.97%	47.22%
13	2	500	49.05	349.33	39.61%	1.38	1.56	-7.39%	78.11%
14	2	1,000	48.62	315.93	40.99%	1.40	1.69	-6.72%	84.35%
15	4	100	38.90	450.41	33.43%	1.77	1.21	-15.14%	30.37%
16	4	500	40.07	206.66	34.08%	1.69	2.64	-12.92%	66.02%
17	4	1,000	39.93	187.64	34.36%	1.71	2.84	-13.35%	71.01%
	GPU backward()								
18	1	10	62.52	218.44	9.64%	-	-	-	-
19	1	50	62.59	199.14	9.82%	-	-	-	-
20	1	100	61.28	185.50	10.09%	-	-	-	-
	Auto Backward()								
21	2	10	41.92	185.40	9.84%	1.49	1.18	0.20%	58.91%
22	2	50	42.40	124.82	9.70%	1.48	1.60	-0.12%	79.77%
23	2	100	42.87	106.03	9.75%	1.43	1.75	-0.34%	87.48%
24	4	10	33.97	133.50	10.37%	1.84	1.64	0.73%	40.91%
25	4	50	32.52	73.58	9.21%	1.92	2.71	-0.61%	67.66%
26	4	100	32.92	61.30	9.86%	1.86	3.03	-0.23%	75.65%
	Manual backward()								
27	2	10	62.51	288.89	9.31%	1.00	0.76	-0.33%	37.81%
28	2	50	62.92	126.68	9.45%	0.99	1.57	-0.37%	78.60%
29	2	100	62.22	107.51	9.84%	0.98	1.73	-0.25%	86.27%
30	4	10	62.90	133.49	9.59%	0.99	1.64	-0.05%	40.91%
31	4	50	61.92	75.29	9.34%	1.01	2.64	-0.48%	66.12%
32	4	100	62.47	61.13	9.92%	0.98	3.03	-0.17%	75.86%

Ref	Nodes	Batch Size	Pre-pro (sec)	Train (sec)	Accuracy	SU-P	SU-T	Acc-Loss	Efficiency
				MNIST					
				CPU SGD Train()					
33	1	100	8.105	168.685	98.27%	-	-	-	-
34	1	1,000	8.234	178.055	98.27%	-	-	-	-
35	1	2,000	7.907	168.386	98.27%	-	-	-	-
36	1	10,000	7.907	174.365	98.27%	-	-	-	-
				Auto SGD Train()					
37	2	100	9.04	215.147	98.03%	0.90	0.78	-0.24%	39.20%
38	2	1,000	8.096	98.849	98.03%	1.02	1.80	-0.24%	90.06%
39	2	2,000	8.015	94.8221	98.03%	0.99	1.78	-0.24%	88.79%
40	2	10,000	8.239	87.058	98.03%	0.96	2.00	-0.24%	100.14%
41	4	100	8.65	166.402	97.40%	0.94	1.01	-0.87%	25.34%
42	4	1,000	7.926	56.324	97.40%	1.04	3.16	-0.87%	79.03%
43	4	2,000	8.053	51.903	97.40%	0.98	3.24	-0.87%	81.11%
44	4	10,000	8.154	44.345	97.40%	0.97	3.93	-0.87%	98.30%
				Manual SGD Train()					
45	2	100	8.79	217.167	98.03%	0.92	0.78	-0.24%	38.84%
46	2	1,000	7.862	105.356	98.03%	1.05	1.69	-0.24%	84.50%
47	2	2,000	7.744	99.46	98.03%	1.02	1.69	-0.24%	84.65%
48	2	10,000	8.04	86.385	98.03%	0.98	2.02	-0.24%	100.92%
49	4	100	7.9	174.102	97.40%	1.03	0.97	-0.87%	24.22%
50	4	1,000	7.805	54.364	97.40%	1.05	3.28	-0.87%	81.88%
51	4	2,000	7.643	52.016	97.40%	1.03	3.24	-0.87%	80.93%
52	4	10,000	7.656	46.797	97.40%	1.03	3.73	-0.87%	93.15%
				CPU backward()					
53	1	100	1.689	411.567	76.14%	-	-	-	-
54	1	500	1.895	398.442	51.83%	-	-	-	-
55	1	1,000	1.656	397.132	39.80%	-	-	-	-
				Auto Backward()					
56	2	100	2.123	204.129	81.93%	0.80	2.02	5.79%	100.81%
57	2	500	2.656	199.237	58.13%	0.71	2.00	6.30%	99.99%
58	2	1,000	2.146	199.933	39.11%	0.77	1.99	-0.69%	99.32%
59	4	100	2.726	102.029	67.86%	0.62	4.03	-8.28%	100.85%
60	4	500	2.883	99.701	39.21%	0.66	4.00	-12.62%	99.91%
61	4	1,000	2.185	100.519	22.41%	0.76	3.95	-17.39%	98.77%
				Manual Backward()					
62	2	100	1.738	206.67	81.83%	0.97	1.99	5.69%	99.57%
63	2	500	1.74	202.01	58.13%	1.09	1.97	6.30%	98.62%
64	2	1,000	1.732	200.661	39.11%	0.96	1.98	-0.69%	98.96%
65	4	100	1.776	102.861	67.69%	0.95	4.00	-8.45%	100.03%
66	4	500	1.767	100.703	39.21%	1.07	3.96	-12.62%	98.92%
67	4	1,000	1.77	101.141	22.41%	0.94	3.93	-17.39%	98.16%

APPENDIX D – TEST RESULTS (MODEL PARALLEL)

Model Complexity Tests

Ref	Model	Nodes	Dataset	GPU	FLS	SLS	TLS	Time	Acc	Efficiency
68	MLP3	1	10K	N	1024	2048	1024	456.29	80.33%	-
69	MLP3	2	10K	N	1024	2048	1024	289.87	80.03%	78.71%
70	MLP3	4	10K	N	1024	2048	1024	226.24	78.83%	50.42%
71	MLP3	1	10K	N	1024	2048	2048	679.69	67.33%	-
72	MLP3	2	10K	N	1024	2048	2048	418.13	83.00%	81.28%
73	MLP3	4	10K	N	1024	2048	2048	300.04	67.56%	56.63%
74	MLP3	1	10K	N	1024	4096	2048	1351.62	78.07%	-
75	MLP3	2	10K	N	1024	4096	2048	764.77	62.57%	88.37%
76	MLP3	4	10K	N	1024	4096	2048	483.04	65.83%	69.95%
77	MLP3	1	10K	N	1024	4096	4096	2230.24	79.23%	-
78	MLP3	2	10K	N	1024	4096	4096	1222.38	70.83%	91.23%
79	MLP3	4	10K	N	1024	4096	4096	728.89	56.70%	76.49%
80	MLP3	1	10K	N	1024	8192	4096	4460.87	55.57%	-
81	MLP3	2	10K	N	1024	8192	4096	2390.47	78.70%	93.31%
82	MLP3	1	10K	N	1024	8192	8192	8082.56	65.10%	-
83	MLP3	2	10K	N	1024	8192	8192	4199.75	70.63%	96.23%

Ref	Model	Nodes	Dataset	GPU	FLS	SLS	TLS	Time	Acc	Efficiency
84	MLP3	1	2K	Y	1024	2048	2048	13.82	74.80%	-
85	MLP3	2	2K	Y	1024	2048	2048	29.38	71.10%	23.52%
86	MLP3	1	2K	Y	1024	4096	2048	26.63	70.80%	-
87	MLP3	2	2K	Y	1024	4096	2048	43.95	69.80%	30.30%
88	MLP3	1	2K	Y	1024	4096	4096	42.86	57.20%	-
89	MLP3	2	2K	Y	1024	4096	4096	51.53	68.70%	41.59%
90	MLP3	1	2K	Y	1024	8192	4096	Error: killed		
91	MLP3	2	2K	Y	1024	8192	4096	68.92	72.40%	#VALUE!
92	MLP3	1	2K	Y	1024	8192	8192	Error: Not enough RAM		
93	MLP3	2	2K	Y	1024	8192	8192	109.2	58.20%	#VALUE!

Model Type Tests

Ref	Model	Nodes	Dataset	GPU	FLS	SLS	TLS	Time	Acc	Acc Loss	Efficiency
94	MLP2	1	2K	Y	1024	2048	-	5.87	67.90%	-	-
95	MLP2	2	2K	Y	1024	2048	-	20.93	72.50%	4.60%	14.02%
96	MLP2	4	2K	Y	1024	2048	-	26.82	64.20%	-3.70%	5.47%
97	MLP3	1	2K	Y	1024	2048	1024	9.64	77.30%	-	-
98	MLP3	2	2K	Y	1024	2048	1024	29.77	66.30%	-11.00%	16.19%
99	MLP3	4	2K	Y	1024	2048	1024	38.13	74.30%	-3.00%	6.32%
100	MLP3	1	2K	Y	1024	2048	1024	9.64	77.30%	-	-
101	MLP3-half	2	2K	Y	1024	2048	1024	22.04	69.30%	-8.00%	21.87%
102	MLP3-half	4	2K	Y	1024	2048	1024	27.1	69.50%	-7.80%	8.89%
103	MLP3	1	2K	Y	1024	2048	1024	9.64	77.30%	-	-
104	MLP3-first	2	2K	Y	1024	2048	1024	14.82	77.90%	0.60%	32.52%
105	MLP3-first	4	2K	Y	1024	2048	1024	17.31	60.20%	-17.10%	13.92%
106	CNN	1	2K	Y	576	200	-	6.09	91.20%	-	-
107	CNN	2	2K	Y	576	200	-	16.59	91.60%	0.40%	18.35%
108	CNN	4	2K	Y	576	200	-	23.36	89.50%	-1.70%	6.52%