PARALLEL DISTRIBUTED DEEP LEARNING ON CLUSTER COMPUTERS

by

Robert Kwan Lee Kennedy

A Thesis Submitted to the Faculty of

The College of Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

Florida Atlantic University

Boca Raton, FL

August 2018

Copyright 2018 by Robert Kwan Lee Kennedy

PARALLEL DISTRIBUTED DEEP LEARNING ON CLUSTER COMPUTERS

by

Robert Kwan Lee Kennedy

This thesis was prepared under the direction of the candidate's thesis advisor, Dr. Taghi M. Khoshgoftaar, Department of Computer and Electrical Engineering and Computer Science, and has been approved by the members of his supervisory committee. It was submitted to the faculty of the College of Engineering and Computer Science and was accepted in partial fulfillment of the requirements for the degree of Master of Science.

SUPERVISORY COMMITTEE:

Taghi M. Khoshgoftaar, Ph.D.
Thesis Advisor

Mehrdad Nojoumian, Ph.D.

KwangSoo Yang, Ph.D.

Nurgun Erdol, Ph.D.
Chair, Department of Computer and Electrical Engineering and Computer Science

Stella N. Batalama, Ph.D.
Dean, The College of Engineering and Computer Science

Khaled Sobhan, Ph.D.
Interim Dean, Graduate College

August 10, 2018
Date

ACKNOWLEDGEMENTS

I would like to thank my parents, Dr. Clarissa Kennedy and Robert Kennedy, for their love and support throughout my studies. I cannot imagine getting this far without it.

I would like to express my sincere gratitude to Dr. Taghi M. Khoshgoftaar, my graduate advisor. His support and mentorship throughout my graduate studies has been invaluable in my progress as a student and researcher. Additionally, I would like to thank Dr. Mehrdad Nojoumian and Dr. KwangSoo Yang for serving on my supervisory committee.

I would like to thank Dr. Joseph Prusa for his help and guidance as I have learned to write research papers and for help with the preparation of my thesis. Additionally, I would also like to acknowledge Joffrey Leevy and all the members of the Data Mining and Machine Learning Laboratory of Florida Atlantic University.

I would like to acknowledge NSF I/UCRC which provided a framework for interaction between FAU faculty and industry, as well as the LexisNexis company for supporting my research assistantship position during my MS studies. I also gratefully acknowledge partial support by the National Science Foundation, under grant number CNS-1427536. Any opinions, findings, and conclusions or recommendations expressed in this thesis are those of the author and do not necessarily reflect the views of the National Science Foundation.

ABSTRACT

Author:	Robert Kwan Lee Kennedy
Title:	Parallel Distributed Deep Learning on Cluster Computers
Institution:	Florida Atlantic University
Thesis Advisor:	Dr. Taghi M. Khoshgoftaar
Degree:	Master of Science
Year:	2018

Deep Learning is an increasingly important subdomain of artificial intelligence. Deep Learning architectures, artificial neural networks characterized by having both a large breadth of neurons and a large depth of layers, benefits from training on Big Data. The size and complexity of the model combined with the size of the training data makes the training procedure very computationally and temporally expensive. Accelerating the training procedure of Deep Learning using cluster computers faces many challenges ranging from distributed optimizers to the large communication overhead specific to a system with off the shelf networking components. In this thesis, we present a novel synchronous data parallel distributed Deep Learning implementation on HPCC Systems, a cluster computer system. We discuss research that has been conducted on the distribution and parallelization of Deep Learning, as well as the concerns relating to cluster environments. Additionally, we provide case studies that evaluate and validate our implementation.

PARALLEL DISTRIBUTED DEEP LEARNING ON CLUSTER COMPUTERS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Machine learning is a field that is rapidly growing in importance and impacting many aspects of our society. Deep learning, at the forefront of machine learning, typically uses very large, very complex artificial neural networks. Like the human brain, these artificial neural networks use connections between neurons and connections between neural layers to perform new tasks and solve new problems. In deep learning, the artificial neural network is generally trained through the process of propagation [38], where the network is given input data and the weights of connections are adjusted based on the calculated error of the networks response to the input data compared to the expected output. Contemporary artificial deep neural networks have been shown to produce state of the art performance after very large networks are trained on vast amounts of data [34]. State of the art deep learning has been applied to many domains from image classification [14] to self-driving vehicles [34] and even outperforming humans in complex competitive games such as Go [40] and Dota 2 (OpenAI Five) [50].

The concept of the artificial neural network has been around since 1943 when McCulloch and Pitts [41] used electrical circuitry to try and explain how neurons in the brain might work. Due to insufficient computing power, the artificial neural network had not proved to be practical in machine learning. By the late 1980s the computing power caught up to the concepts and a more efficient backpropagation algorithm was introduced by Lecun et al. [38]. Since then, the modern artificial neural network has become prominent in the machine learning domain. In 2012 one of the first successful deep learning networks, called AlexNet [35], significantly outperformed traditional

machine learning techniques on image classification. This was a significant turning point where it became clear deep learning could greatly outperform other more traditional machine learning techniques. Since then, increases in dataset sizes and neural network architectural sizes and complexities have produced cutting edge deep neural networks that have continually outperformed their predecessors [30, 40, 14, 34, 50]. As dataset sizes increase and the neural network architectures grow more complex, the computational power and memory demands of training also grow. The demand has grown so quickly that state of the art Deep Learning is being done on large supercomputers and cluster computer systems with thousands of processors. Training neural networks in this type of environment requires a new approach. Dean et al. [18] introduced new techniques and frameworks for training distributed models for speech recognition and computer vision that handled the parallelism and communication between the nodes but was on systems that are generally unavailable to most [48]. An approach utilizing more readily available systems was introduced by Coates et al. [17]. Their system, a cluster of readily available GPUs and based off on Commodity Off-The-Shelf High-Performance Computing technology, is much more readily available and produced similar results to the more exclusive ones in [18]. The continued research and development of cutting edge Deep Learning will require the use of high performance cluster computers and distributed learning techniques that run on them.

1.1 MOTIVATION

Training neural networks effectively and efficiently is an important component of Deep Learning. Large neural networks can consist of dozens, hundreds or even thousands of layers each with thousands of artificial neurons. Depending on the network's architecture, each of these neurons is connected to a large number of other neurons, where each connection has a trainable weight parameter that determines how the network responds to input signals. In context of this thesis, the effective training of these large

complex networks is accomplished through the use of the computationally expensive process of backpropagation, see Section 2.1. Additionally, neural networks benefit from training on very large amounts of data, typically the larger the better [18, 65]. For example, the ImageNet database AlexNet was trained on consist of roughly 1.2 million images. Thus, massively powerful computers are required to efficiently train large neural networks on large amounts of data.

To keep up with the computational demands, training neural networks in parallel across cluster computer systems or shared memory supercomputer is an active area of research, see Chapter 2. As the models and datasets grow, it is possible to not only improve the hardware of the computer (such as more memory, faster CPU, faster GPU, etc...) but to also increase the number of computers simultaneously training a single model. Possible parallel methods for training a neural network include splitting the data across multiple computational nodes and splitting the neural network itself across multiple nodes, known as Data Parallelism and Model Parallelism respectively. Each have their benefits, but both aim to reduce training time by spreading out the required computations across many different computers. Additionally even the distributed systems themselves are an important factor when trying to parallelize the neural network training [18, 17]. A supercomputer can have thousands of individual processors that are connected via a specialized high speed (i.e. a shared memory supercomputer). However, a cluster computer system, such as the one used in this thesis (HPCC Systems), is comprised of multiple individual computers, each with their own processor, memory, and disk. They are connected together via high speed networking interfaces to form the cluster. Both of these systems require not only specialized Deep Learning methods, but also specialized software to handle the synchronization across nodes (CPU, GPU, or individual computers), the communication between nodes, and the parallelization of the data and the process appropriate to the specific distributed system. This thesis uses High-Performance Computing Cluster

(HPCC) Systems as the cluster computing environment. It provides the necessary logistics to provide an effective and efficient environment for distributed Deep Learning by leveraging TensorFlow, a popular open source deep learning library.

The goal of this thesis's work is to implement a distributed parallel neural network training runtime on the HPCC Systems platform. We intend to combine the big data handling capabilities of the HPCC cluster system with widely used, open source deep learning libraries to provide a platform for future development of cutting edge deep neural networks.

1.2 CONTRIBUTIONS

HPCC Systems and TensorFlow are both currently available to researchers as open source software. Prior to this thesis's work, HPCC Systems did not have the facility to perform Deep Learning in a distributed and parallel fashion. This thesis presents a novel distributed parallel stochastic gradient descent method for HPCC Systems, using TensorFlow. We leverage HPCC's cluster environment, compiler, and specialized programming language to provide the parallelization of the data and learning processes. In addition, we leverage open source libraries, such as TensorFlow and Keras, to abstract Deep Learning models and to execute the localized training procedures on each node of the cluster. The contributions of this thesis can be summarized as follows:

1. An open source implementation, using ECL (HPCC) and TensorFlow, of a synchronous data parallel method (parallel stochastic gradient descent) to provide HPCC Systems distributed Deep Learning capabilities.

2. A study of the impact of data size and cluster size on neural network training time using a synchronous data parallel optimizing runtime on HPCC Systems.

3. A study of the impact of data size and cluster size on neural network model

performance using a synchronous data parallel optimizing runtime on class imbalanced Medicare data on HPCC Systems.

1.3 ORGANIZATION

The remainder of this thesis is organized into the following chapters:

- Chapter 2 provides background on parallel neural network optimization, strategies, and paradigms, and other related work.
- Chapter 3 provides details on the software libraries used in the implementation, HPCC Systems, and the methodology of the implementation.
- Chapter 4 presents two case studies validating the implementation described in Chapter 3
- Chapter 5 presents the conclusions of our case studies and discussion of future work.

CHAPTER 2

RELATED WORK

The focus of this thesis is the implementation and validation of distributed parallel neural network training using existing and proven paradigms on HPCC Systems. This chapter provides an overview of neural networks and neural network optimization, details of parallelization strategies, parallel training paradigms, and select parallel optimization algorithms. A detailed review of HPCC Systems and an overview of the neural network frameworks used in the implementation are also provided.

2.1 NEURAL NETWORK TRAINING

The Backpropagation Algorithm is the main method for training artificial neural networks, specifically, deep neural networks [36]. Backpropagation is comprised of two steps, a forward and backward pass through the model from input to output and vice versa, respectively. On the forward pass, a number of training examples, depending on the variation of SGD being used, are fed into the input of the model and the output's loss is calculated. In the second step, a backward pass calculates the gradient of each weight with respect to the loss of all neurons in a layer. The backward calculations propagate to each successive layer, stopping at the first hidden layer. The calculation of the gradient error is a direct application of the chain rule [60].

Neural networks are numerically described as a highly non-convex optimization problems in high dimensional space. Stochastic Gradient Descent (SGD) and its derivatives are the most commonly used methods for solving this problem, i.e. training

a neural network [8]. For the purposes of this paper, there are three main variations of stochastic gradient descent: stochastic, batch, and mini-batch stochastic gradient descent. Standard stochastic gradient descent uses singular data points when performing gradient descent, batch uses use the entire dataset at once for each iteration of gradient descent, and mini-batch is a sort of hybrid where a small batch size is used for each gradient descent iteration. One of the main strengths of SGD is it is relatively easy to implement and works well for problems that have a large number of training examples, such as training artificial neural networks. However, one key disadvantage to using SGD methods is that in order to work optimally, the methods need to be tuned. There are multiple hyper-parameters to the method itself, and each one needs to be tuned on the problem at hand. This is an issue because if the problem at hand is not well known, or even misjudged, the tuning of the method becomes difficult. Another disadvantage, which is a part of this papers motivation, is that SGD methods are inherently iterative and subsequently are difficult to parallelize.

Artificial neural networks are typically trained in a supervised fashion. In machine learning, supervised learning is the process of training a model, such as an artificial neural network, using labeled examples. Each training example Z is represented by a pair (x,y). x denotes an arbitrary input and y denotes a scalar output. A cost, or loss, function $\mathrm{c}(\hat{y},\ y)$ quantifies the loss of the neural network by measuring the difference between a predicted value y and the actual, or truth, value of $\hat{y}$. Function $f\colon\ X \rightarrow\ Y$, where X is the input space and Y is the output space, represents the network, and is parametrized by weight vector w. We aim to find the function f that minimizes the loss function $\mathrm{c}(\hat{y},\ y)$. Gradient descent has been shown to minimize expected loss of f [60] and exactly calculates the gradient of f and is computationally intensive. Stochastic Gradient Descent, on the other hand, is an iterative method to approximate the gradient using backpropagation [39]. The approximation reduces the required computation and time. Since SGD is an approximation, it is not guaranteed

to find the global loss minimum for the model's error due to the loss function's non-convexity. However, it has been shown the loss converges similarly to gradient descent [8, 67], i.e. the training process of the model increases its performance every iteration. The approximation of the gradient, on a single data point basis, generates noise as the algorithm optimizes the model. This noise however, affects the speed of convergence. To combat this, SGD uses a learning rate, that is selected during implementation. Changing the learning rate will in effect speed up or slow down the convergence speed on a given dataset. Careful selection of the learning rate is critical in the training process. A learning rate value that is too high will converge very quickly, but can overshoot, even oscillate away from local minima; too small of a learning rate can cause the network to train slowly and underfit.

Since SGD is a first order approximation, the rate of change of the gradient is not calculated. A slight variation of the SGD, called Stochastic Gradient Descent with Momentum, introduces an adaptive learning rate to optimize the convergence [57]. A momentum term is added into the weight update where the current weight update is dependent on the previous updates. It improves the optimizers convergence speed and has been shown to avoid getting stuck if the error curve flattens [67], i.e. the optimizer converges to a flattened part of the loss curve rather than a local or global minima.

2.2 PARALLELIZATION STRATEGIES

Training a neural network in parallel has two main factors that contribute to the training time. First is the communication cost between nodes and second is the computational time cost. Increasing the number of parallel nodes reduces the system's computational time but at the expense of high communication costs. A balance between the two factors results in decreased training time and is dependent on the architecture of the physical system as well as the methodology of splitting up the

computations.

There are 6 different dimensions of parallelization for neural networks introduced by Nordstrom et al. [49], each with increasing communication costs. In the simplest dimension, training session parallelism, there exists virtually no communication costs. Each node in the system gets an entire copy of the dataset and model, each initialized with different weights. Each of the nodes trains a model and at the end of training, the model that performs the highest is selected. This technique, although rather trivial in concept, can be useful because training neural networks can have the propensity to get stuck in local minima. The multiple, initialized models aim to widen the search to hopefully find a global minimum [51]. Another advantage to this approach is its parallelism is unbound, i.e. the increase in nodes and the decrease in training time (for that many different models) is nearly perfectly linear. The next dimension, Training Example Parallelism, also known as Data parallelism, see Section 2.3, splits the training data onto multiple nodes. Each node then computes on a smaller data size, reducing training time. This approach is examined in more detail in the coming sections. The final dimension that is suitable for a cluster computer environment is Node Parallelism, similar to model parallelism detailed in Section 2.4.

One of the most challenging aspects of large scale distributed machine learning is the parallelization of neural network training when using Stochastic Gradient Descent [61], and any derivative. There have been many studies that attempt to parallelize SGD [51, 18, 81, 79, 49]. Many of these are designed to use some form of a shared memory computer. Often referred to as a supercomputer, these systems have the benefit of tightly controlled communications between different processing units, either CPU's, CPU cores, or GPU's. However, this thesis' work focuses on parallelization methods that are well suited for use on a cluster computer. The main difference being the communication between nodes is not a specialized serial bus, but rather a standard ethernet connection. This presents a high cost of communica-

tion due to the slower ethernet connections and standard networking protocols. The network itself introduces a bottleneck. Consider the scenario when all workers need to communicate their updates for aggregation concurrently, the network can become easily saturated. leading to increased training time. Careful consideration of the communication strategy of the parallelization is critical [64].

2.3 DATA PARALLELISM

In general, data parallelism is the paradigm, during the training phase of creating a machine learning algorithm, where the data used for training is partitioned and distributed evenly across several nodes of a system, along with a copy of the initialized, untrained model is also distributed to each node or worker. All workers will concurrently train on their partition of the data and at the end of the training processes, the models are aggregated in some way to have the system produce a model that was trained on the entire dataset; effectively reducing the amount of work required by each node by a factor of how many workers are used. This method is one of the oldest practical implementations of training an artificial neural network [80]. There are several different data parallel approaches, mainly differing by how the workers' contributions are aggregated. They can be categorized into two main groups: Synchronous and Asynchronous Data Parallelism.

2.3.1 Synchronous Data Parallelism

Synchronous Data Parallelism, as its name implies, is a training paradigm where the training steps are executed in a synchronous and sequential way. It is identifiable by a series of locking mechanism during the training runtime. The synchrony ensures model consistency between training steps. Using mini-batch gradient descent as an example on a single machine, the dataset is partitioned into a series of mini-batches where each mini-batch is consumed in sequence until all the data has been consumed

(see Algorithm 1). Then the weights for each mini-batch are averaged according to this formula which sums of all the mini-batch's gradients divided by the number of mini-batches. The locking mechanism in this case is the aggregation after each pass through the data and makes mini-batch stochastic gradient descent synchronous in nature. Even on a single machine, mini-batch SGD has been shown to optimize the training time [36].

Algorithm 1 Mini-batch Stochastic Gradient Descent with Backpropagation

```
1: for epochs do
2:     for t=0 to B/D do
3:         Sample B elements from D
4:         Compute Gradient with respect to B
5:         Update Network Weights
6:     end for
7: end for
```

A distributed adaptation of mini-batch SGD, where each node in a system computes on a single mini-batch, is also a synchronous approach. Each worker calculates its weight updates on its partition of data and a master node, or parameter server, aggregates the weights. The locking mechanism comes from the fact that after each pass through of the data, the parameter server has to aggregate the weights before continuing onto the next iteration. If each worker in the system takes exactly the same time to train, the synchrony is not an issue. However, if one or more workers takes longer than the rest, the synchrony becomes a significant bottleneck in the training time. The workers can have varying training times for several reasons such as a cluster system with heterogeneous hardware, slightly different data partition sizes, network latencies, etc. Careful consideration needs to be taken to minimize the chances for this type of bottleneck. Furthermore, this approach requires parameters updates to happen on each mini-batch iteration. This results in a large communication costs in a cluster computer.

Parallel Stochastic Gradient Descent

Designing a parallelization method that considers the high communication costs found in cluster computers has been extensively studied [79, 51, 13]. Parallel SGD, shown in Algorithm 2 and Algorithm 3, introduced by Zinkevich et al. [81] is one such technique and can be viewed at as an improvement on model averaging. Model averaging convergence is dependent on the degree of convexity as a result of regularization. However, regularization decreases with increase in data size which makes it less useful in practice, especially since data sizes continue to grow. Parallel SGD improves upon this by combining the benefits of low network communication of model averaging and that of online learning [81, 46]. In practice, as well as in the contributions of this paper, Parallel SGD is implemented by first splitting the dataset into N parts, where N is the number of nodes. Second, a copy of the model, staring from the same weight initialization, is distributed to each node. Next, each node trains on its partition of data for as many epochs as needed. Importantly, the data is shuffled between each epoch on each node and uses SGD, or in the case of this paper, a mini-batch SGD. At the end of the training process, all node weight contributions are summed. Specifically, each node's weight updates are divided by the number of nodes, see Algorithm 3. Since the weights are aggregated only at the end, the communication cost is only a constant. The time required to train is only dependent on the size of the data and the number of epochs selected for training which makes this an ideal method for systems with communication bottlenecks, such as in cluster computing.

Algorithm 2 SGD $(\{c^1, ..., c^m\}, T, \eta, w_0)$

for t=1 **to** T **do**
  Draw $j \in \{1...m\}$ uniformly at random
  $w_t \leftarrow w_{t-1} - \eta \partial_w c^j(w_{t-1})$
end for
return w_T

Algorithm 3 ParallelSGD $(\{c^1, ..., c^m\}, T, \eta, w_0, k)$

for all $i \in \{1, ..., k\}$ **parallel do**
  $v_i = SGD(\{c^1, ..., c^m\}, T, \eta, w_0)$ on each compute node
end for
Aggregate from all nodes $v = \frac{1}{k} \sum_{i=1}^{k} v_i$
return v

2.3.2 Asynchronous Data Parallelism

Asynchronous data parallelism methods try and remove some of the locking mechanisms inherent in a synchronous parallelism. Recall that in a synchronous setting, the aggregation of all workers or all mini-batches needs to occur before the next epoch and at the beginning of each epoch, all workers are starting from the same model parameters hosted in the parameter server. Removing this locking mechanism would allow for stale workers to continue to train, further optimizing the system and further reducing training time. After distributing the data to the worker, each worker, in parallel, processes a mini-batch independently of the others. First the worker fetches the most up to date parameters from the parameter server. Then it calculates the gradient using its mini-batch with respect to these fetched parameters. Lastly, the worker sends the weight updates to the parameter server, which then applies them to the master model. The asynchrony is derived from the independent worker processes. Formally, this is called Asynchronous Stochastic Gradient Descent and was introduced by Dean et al. [18]. For completeness, we provide the algorithm for an example asynchronous optimizer; called *Downpour SGD*, shown in Algorithm 4 which was introduced by Dean et al. [18]. What makes this fundamentally not synchronous is a model update can occur while other workers are still processing, making the parameters they are processing outdated. The number of model updates that have occurred while processing old parameters, or to what degree the parameters are outdated, is referred to as its staleness. In practice, the workers gradient updates are typically computed from stale parameters [13]. The negative effects of the staleness

have been the focus of recent papers [55] which show asynchronous work under strong assumptions that might not be true in practice.

Algorithm 4 DownpourSGD($\alpha, n_{fetch}, n_{push}$)

```
procedure StartAsyncFetchParameters(parameters)
    parameters ← GetParametersFromParameterServer()
end procedure

procedure StartAsyncPushGradients(accruedgradients)
    SendGradientsToParameterServer(accruedgradients)
    accruedgradients ← 0
end procedure

Main
    global parameters, accruedgradients
    step ← 0
    accruedgradients ← 0
    while true do
        if (step mod n_fetch) == 0 then
            StartAsyncFetchParameters(parameters)
            data ← GetNextMiniBatch()
            gradient ← ComputeGradient(parameters, data)
            accruedGradients ← accruedGradients + gradient
            parameters ← parameters − α * gradient
        end if
        if (step mod n_push) == 0 then
            StartAsyncPushGradients(parameters)
            step ← step + 1
        end if
    end while
End
```

2.4 MODEL PARALLELISM

In general, model parallelism is a paradigm where a machine learning model is split across several nodes for training. Each partition of the model then has its training computed by the partition's respective node. Adhering strictly to the model parallelism paradigm, the entire training dataset is consumed by the node for each model partition. Thus, rather counterintuitively, the data is copied and distributed

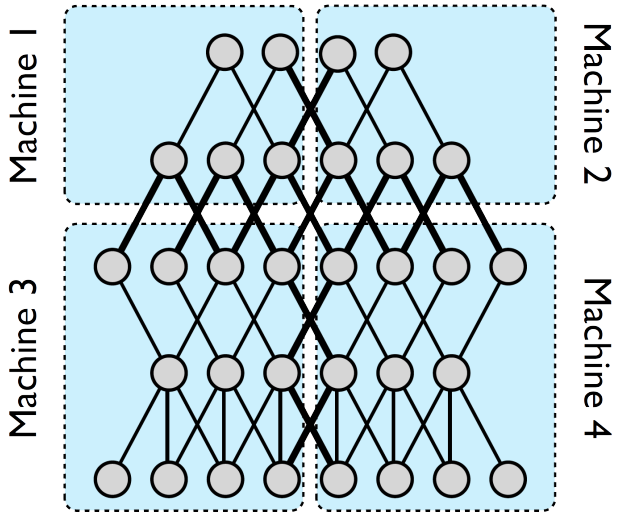


Figure 2.1: Originally from Dean et al.[18], depicts a five layer neural network that is distributed across four nodes (the blue squares). Only the bold edges, ones that span between nodes, need to have their state communicated.

to all nodes. Each node can either be individual computers, separate CPU's, CPU cores/threads, GPU devices, or other computational resources. However, due to how forward and backward propagation works, each section of the partition needs to communicate with its adjoining parts in other partitions. Specifically in neural networks, the node pairs that lie on partition boundaries need to communicate across the boundary, see Figure 2.1.

Current large neural networks can have a large number of nodes, layers, and connections between layers, and can be big enough to be larger than what can fit in memory. Model parallelism alleviates this issue by splitting the larger than memory model across multiple nodes. These, and models that can fit in memory, benefit from access to more memory and computational resources up until the communication costs across partition boundaries becomes too high [18]. As a result, the performance

benefits of using model parallelism is dependent on the architecture of the model and the communication latency between nodes. A shared memory system (e.g. a super-computer) would scale well due to its low latency between its processors.

2.5 PARALLEL PROCESSING SYSTEMS

In the mid-1980's, new parallel relational database architectures were created to increase performance when analyzing large amounts of data. Teradata and the Gamma projects started a new architectural paradigm using multiple commodity computers in parallel [66]. Partitioning a database's tables and SQL queries onto different nodes achieves this parallelism, and this architecture has been used as the basis of nearly all parallel relational database systems since. Though this approach performs well with large data that is structured that can easily be constrained to a relational database, new processing paradigms using flexible data models were needed to handle the increasing data sizes that were primarily unstructured [56, 20, 11]. As a result, several different solutions were developed, including Google MapReduce, Hadoop, and HPCC Systems [19, 45, 25].

2.5.1 MapReduce Architectures

MapReduce is a programming model and implementation that parallelizes computations across a cluster of computers. It gets its name from the two main computation tasks a user specifies, a map function and a reduce function that is based off of a set of input and output of key/value pairs. The map function takes an input pair and creates an intermediate key/value pair and are combined with other matching input pairs by the framework. The reduce function uses the intermediate pairs to perform aggregation for the output. The MapReduce process is inherently parallel since the model handles partitioning the data, communication between the nodes, and scheduling and executing jobs without direct user input. Sequential series of

MapReduce operations are used to perform more complex operations.

Hadoop is an open source project that implements Google MapReduce architecture; it is functionally the same as MapReduce but is written in a different underlying language [75] and is designed to run on a cluster of commodity computers running on Linux. Hadoop's implementation provides a distributed data processing, scheduling, execution environment, and framework for creating MapReduce jobs using a master and slave architecture. It provides its own distributed file system based off of Google MapReduce implementation that also uses the same master/slave architecture for file management.

2.5.2 Limitations of MapReduce

The MapReduce model can be used for performing ETL (Extract, Transform, Load) tasks on large data, aggregating large unstructured, data and other basic data processing. However, many data processing tasks are not easily completed with the MapReduce's key/value pair and group by aggregation. Users are forced to conform their logic to the MapReduce paradigm to achieve parallelism such as projections and selections [11]. In addition, more complex runtimes also need to conform to the MapReduce paradigm to utilize its parallelism. It is required to use sequences of MapReduce functions to perform more complex calculations which ultimately adds significant overhead and limits re-usability [19, 11]. However, these limitations have been addressed by some high-level languages such as Sawzall and Yahoo Pig. These languages allow the user to adhere to a Dataflow paradigm and use standard data processing functions, beyond just aggregation, by compiling down into standard MapReduce functions, without having to implement specific functions MapReduce. This also provides additional reusability and added optimization. In contrast to MapReduce, the Dataflow [58, 27] model can be represented as a directed graph that represents the flow of data and transformations of that data in a program, see Figure 2.2 for

an HPCC specific example. Nodes represent program operations and the edges between nodes represent queues. However, the high-level languages and their provided functions still rely on the underlying MapReduce model and thus are still limited to the execution model. Since MapReduce is just a programming model for data processing, it does not provide any facility for data warehousing like a traditional relational DBMS [66]. Other accompanying platforms, combined with MapReduce, are required. This too has been addressed. Google developed BigTable [12], and Hadoop has HBase and Hive [25, 75]. These combine with the data processing provided by MapReduce to provide a system with benefits such as efficient querying of large data sets, low latency random access of structured data, and data warehousing as found in DBMS [45, 66].

2.6 HPCC SYSTEMS

High-Performance Computing Cluster (HPCC) Systems is an open source software platform developed in 2000, and currently in use and maintained by LexisNexis, as a solution for data-intensive computing. HPCC is an integrated system environment that excels at ETL operations, complex analytics, and provides efficient querying of large data sets by a large number of uses with its own data-centric parallel processing language called ECL (Enterprise Control Language) [45]. It combines the benefits of MapReduce or Hadoop, its high-level languages, and other additions as described in the previous section such as HBase and Hive. The system is comprised of newly developed system software and middleware components that are layered on top of the Linux operating system, also open source. This combination provides an execution environment and distributed file system that is required by data-intensive computing.

Data-intensive computing is defined as software applications that are limited by I/O (input and output) or ones that need to process large amounts of data [31, 24], these types of applications spend a large percentage of time processing and moving

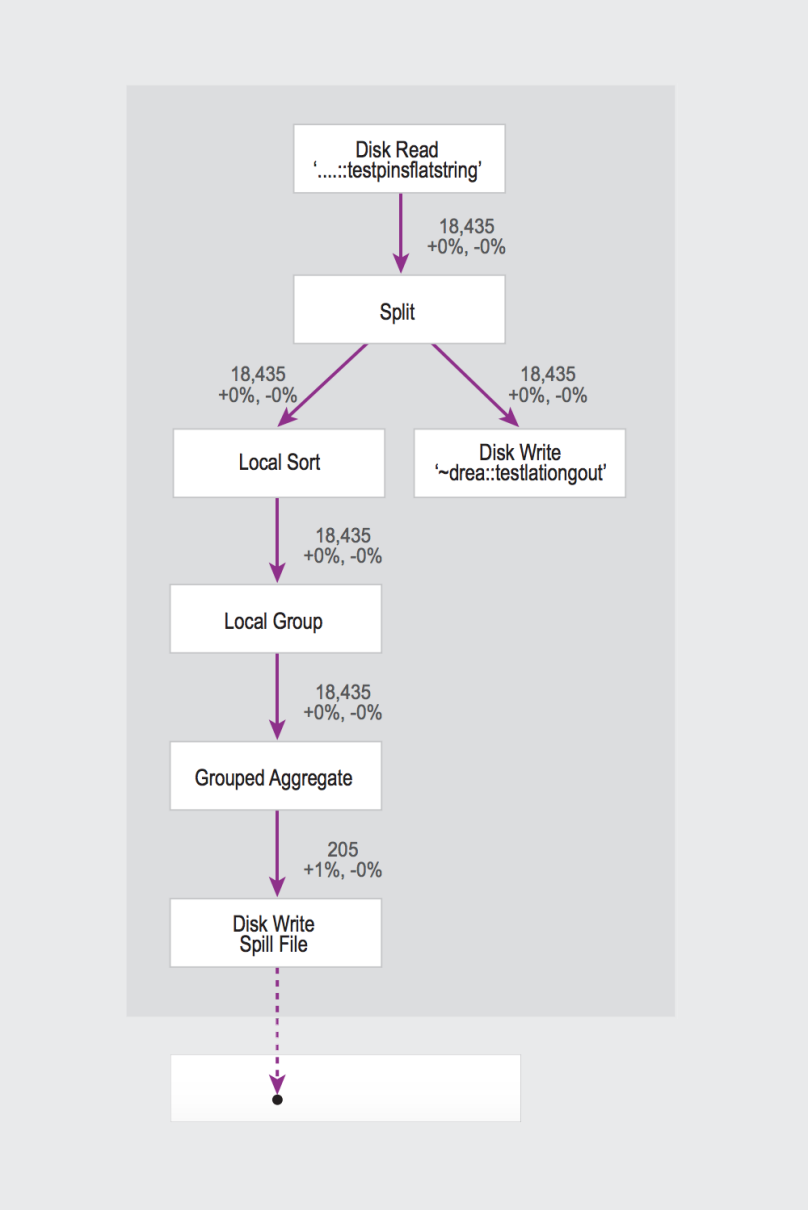


Figure 2.2: Originally from [45], this figure denotes a dataflow diagram. Specifically, an HPCC record dataflow diagram showing the flow during a sort.

around data; in this case, between nodes in a cluster. A distributed implementation of neural network training would fall under this definition as it needs to both move around data between nodes and process large data covered in Section 2.2. HPCC Systems was developed to be a data-intensive computing system, rather than other types of computing. HPCC collocates the data and programs, on each node, to reduce the movement of the data which increases performance and parallelism[45, 24]. It uses a high level language abstraction (ECL) to make the system machine independent which allows for deployment on different types of systems [10]. As a result, the underlying hardware on which an HPCC System is deployed can be scaled linearly as the data and processing requirements grow [45]. HPCC is designed to run on commodity computing clusters (cluster computer) with each node running a Linux operating system. A cluster computer is a group of individual computers each connected via a network. This is in contrast to a supercomputer which for the purposes of this paper is a system with multiple processing units using specialized hardware to allow for RAM memory to be shared. A benefit to the cluster computer is it is trivial to add another computer to system, essentially just plug it in. In the case of cloud computing such as Amazon Web Services (AWS) or Microsoft's Azure, just an additional computer instance needs to be turned on [44, 73, 74].

HPCC Systems platform excels at both ETL tasks and analytics using its own programing language, ECL. ETL operations are used to read data from external sources, cleaning and pre-processing the data so it is in a consistent format for use in a system, and loading that data onto the internal database [45]. ECL adheres to the Dataflow paradigm and similar to many other approaches, HPCC is designed to use a cluster of commodity computers. HPCC Systems consists of two cluster types, a rapid data delivery system known as Roxie, and a data refinery known as Thor. Roxie [45] is used for high throughput delivery of data that is prepared by Thor and is out of scope of this paper.

The Thor cluster [45] uses a master/slave design with a single master Thor process and multiple slave Thor processes. HPCC Systems is designed to run a cluster of Linux based computers which can be configured to have either single Thor processes per physical node, or multiple Thor processes per physical node and can scale to thousands of processes across thousands of physical nodes[1], see Figure 2.3. A Thor cluster uses its own distributed file system (DFS). It can include data from many different types of sources and different formats such as, CSV, XML, numeric, strings, and even binary large objects (BLOBS). The data is in a record format which is different than the block type found in a MapReduce based system. A dataset consists of multiple records, each of which can be different value types and can be variable in length as well as having nested records. When a dataset is uploaded to a Thor cluster, it is evenly split across all of its processes no matter how big the data; i.e. a 400-node system would have the dataset distributed across all nodes into 400 equal parts. It uses the nodes operating system for physical file storage and does not separate individual records. The master Thor node is responsible for monitoring each slave's processes, handles the network I/O for distributing data and distributes ECL code (which is compiled into C++ for execution) to each slave node for execution.

2.7 ECL LANGUAGE

HPCC Systems uses its own programming language called ECL [45]. ECL is an implicitly parallel declarative language that compiles into C++ and is based on the Dataflow model. As with all declarative languages, execution is not determined by the order of the statements, but rather the sequence of operations on the data defined by the Dataflow, as described in previous sections. The high-level nature of the language lends itself high reusability and extensibility and enables users to efficiently generate complex runtimes. HPCC Systems includes an IDE for ECL and an opti-

[1]For the purpose of this paper, the configured system has one Thor processes per physical node and the term node is used interchangeably with the term process.

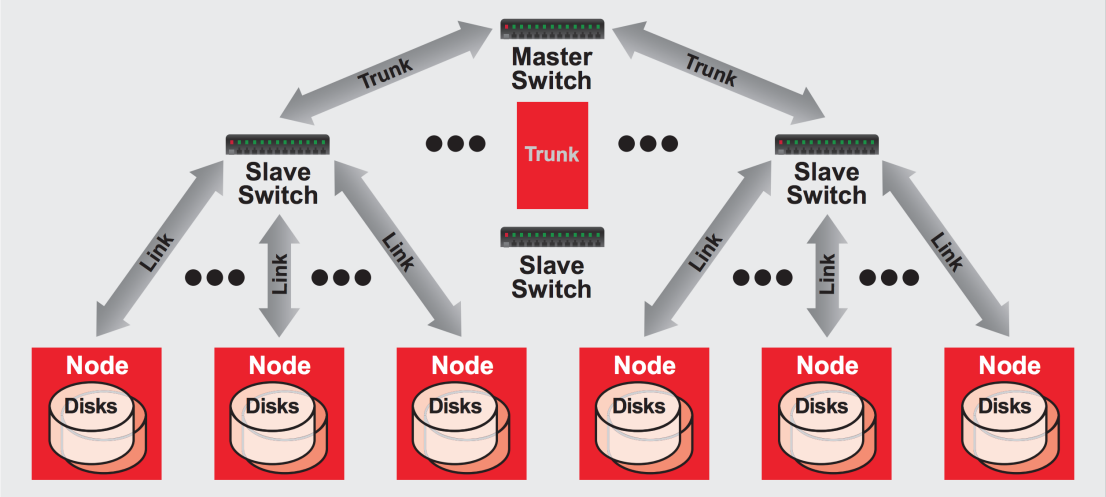


Figure 2.3: Visualization of an example HPCC Systems cluster.

mized compiler that compiles ECL into C++, optimized for distributed parallelism across the Thor nodes. As a result, ECL is easily extended by any standard C++ libraries or .DLL's.

ECL has extensive capabilities for data management, filtering, transformation, and has built in functions that facilitate user defined transformations functions that transform records. Some of the built-in functions include: PROJECT, ITERATE, ROLLUP, JOIN, COMBINE, FETCH, NORMALIZE, DENORMALIZE, and PROCESS. In addition, users can embed C++ code directly in line with ECL. Users are also able to embed other languages, through HPCC plugins, such as R, JavaScript, and Python, and any of their available libraries. This paper's main contributions rely on the combination of ECL and parallel embeddings of other languages and libraries. The Thor cluster allows for these ECL capabilities to function on all of the records distributed across the system or to function on each data partition on each node locally. This is either explicitly defined by the user or implicitly defined by which function is used. This differentiates HPCC from MapReduce where in a MapReduce setting each operation is done locally only. For example, a local sort in HPCC would sort the records on each node individually, without regard to other parts of the dataset on other nodes, a global sort would be performed on the entire distributed dataset

and HPCC Systems handles all the data transfers between the nodes to accomplish the sort.

2.8 TENSORFLOW

TensorFlow is an open source programming library created by Google [1]. Like ECL, TensorFlow implements the dataflow model that represents programs as directed graphs. Fundamentally, TensorFlow is a symbolic math library that is primarily used for building and training artificial neural networks. It not only used for development and training of neural network models but also the deployment of the models for practical use.

The Google Brain team created TensorFlow to solve their need for very large-scale models. Its predecessor is DistBelief [18], a software library used for training neural networks internally at Google. DistBelief was designed for large scale distributed training of neural networks which had two algorithms that were asynchronous in nature and were well suited for a shared memory system. DistBelief, and now TensorFlow, has been used at Google for a wide range of research problems such as unsupervised learning, language representation, image recognition, speech recognition, models for playing Go [40], and reinforcement learning among others [35, 21, 68, 77, 26, 29]. They have deployed models for practical use in widely used Google products like YouTube, Google Maps, and others [1]. TensorFlow is the second generation machine learning library and improved DistBelief by being more flexible, more performant, trains a wider gamut of model all while being used on heterogeneous systems. TensorFlow is not only the standard machine learning library in use at Google, it also is one of the most popular deep learning libraries in the community [22, c.2017]. Thus, it is the machine learning library used our work.

TensorFlow is a Python library, written in Python and C, and can be imported and used as any other Python library. There are other supporting libraries, which are

included in installation. A TensorFlow example is provided below for completeness. This TensorFlow logic defines an multi-layer perceptron (MLP) used for classifying the MNIST dataset.

```
n_hidden_1 = 256 # 1st layer number of neurons
n_hidden_2 = 256 # 2nd layer number of neurons
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)

# tf Graph input
X = tf.placeholder("float", [None, n_input])
Y = tf.placeholder("float", [None, n_classes])

# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}

# Create model
def multilayer_perceptron(x):
    # Hidden fully connected layer with 256 neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Hidden fully connected layer with 256 neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
```

```
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer

# Construct model
logits = multilayer_perceptron(X)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)
# Initializing the variables
init = tf.global_variables_initializer()
```

Code Example 2.1: TensorFlow MLP

CHAPTER 3
METHODOLOGY

This chapter provides information on the system our code is implemented on, the implementation itself, the tools and hardware used, and how the validation experiments were conducted. The following sections provide details of the distributed system, the hardware the system uses, additional software, and tools used for the implementation. Additionally, we provide details of the validation of our implementation as well as discuss the performance metrics used in validation.

The primary goal of this thesis is to implement distributed parallel deep learning on HPCC Systems. To accomplish this goal, we used HPCC Systems ability to run Python code, TensorFlow as our machine learning library, and Keras as a higher-level interface to TensorFlow. We based our implementation off of the Parallel Stochastic Gradient Descent algorithm using data parallelism.

3.1 SYSTEM ENVIRONMENT

We use HPCC Systems as our main environment for distributed training, which has been shown to perform and scale well with big data [47]. It handles the nodes, their execution of code, the communication between nodes, etc. The implementation is designed so that an end user will be able to quickly and easily train a neural network on any HPCC Systems in a distributed and parallel fashion. Something HPCC was previously incapable of. Following the data parallel paradigm, the main dataset is partitioned into smaller parts. These partitions are then placed onto individual nodes for localized computation, or in this case, neural network training, see Section 2.3.

After the training is completed on each node the results from each worker node is then transmitted to a master node for aggregation. This work-flow is accomplished through use of the HPCC Systems. Writing code in ECL, the HPCC language, to partition and distribute the data is trivial and is outlined in Section 2.7. HPCC handles all of the networking, and coordination between the master node and slave nodes intrinsically. Once the data is sent to the worker nodes, it needs to be used to train a neural network. This is where TensorFlow is used.

TensorFlow, see Section 2.8, can be looked at as the neural network training engine. Since HPCC provides the ability to write in different languages, such as Python, alongside ECL code, it is possible for ECL code to work alongside TensorFlow code, as it is a Python library. Specifically, the Python code is written as complete functions (either as TensorFlow functions, or other Python functions using any library as needed), rather than inline operations. ECL then has the ability to call these Python functions while passing in any required parameters. TensorFlow was chosen over other libraries because it is one of the most prominent deep learning libraries. It provides flexibility in building the neural network architecture as well as flexibility in how to optimize and train the models. Being open source also is a benefit. This allows for a large community of development support and the library itself is constantly being developed and improved by a large number of people. Additionally, we chose to use another library that is widely used, called Keras, that would allow us to easily change which neural network engine to use without any modification to our implementation.

Keras is a high-level neural network API (applications programming interface) written as a library for Python, and was created with a focus on neural network development speed. This API interfaces with multiple different machine learning libraries that act as the backend, TensorFlow being one of them. This not only speeds up the processes of building models and training models, but allows to easily switch

to other backends, i.e. this thesis' implementation is not limited by TensorFlow[1]. Our contribution is combination of ECL code and Python code running on HPCC Systems, where the Python code is used for local worker model training and ECL code is used for distributing the data and worker code as well as executing the worker code and aggregating the workers results. The Python code, or functions, is referred to as a "pyembed", short for "Python embedded code".

3.2 SYSTEM HARDWARE

Case studies investigating our implementation use multiple size clusters running HPCC Systems. We use multiple different size clusters, using identical node hardware, to provide comprehensive validation and analysis of how our implementation scales on HPCC with respect to data size, number of nodes in the cluster, and neural network type. All clusters used in this thesis were hosted as cloud compute instances on Microsoft's Azure platform. Cloud based virtual machines are ideal for researching cluster computing. The availability of each instance is immediate, the costs are relatively low and charged by the minute, and scalability from a single node to many nodes is logistically trivial [43, 78]. Each node, Thor node or worker, in the system is represented by an individual cloud computer running Ubuntu (version 16.04 LTS). Careful consideration was taken to minimize any bias when comparing between clusters. Each node, for all clusters and validation experiments, is a DS1_V2 standard general purpose instance, each with 1 CPU (1 core), 3.5 gigabytes of RAM memory, and solid-state drives [42]. All instances were in the same Azure region (meaning they were in the same cloud data center) and shared their own network and subnet. Additionally, each instance was created from the same virtual image using an automated tool to ensure that every and all nodes used for this paper were identical to allow for comparisons between clusters sizes.

[1]For the purposes of this paper, the terms Keras code, TensorFlow code and Python code can be used interchangeably

We used the default configuration manager settings when creating our HPCC clusters, with one Thor slave node per physical node. Albeit the "physical nodes" are cloud based virtual machines, Microsoft Azure provides the virtual machine instance as if it were its own computer, including its own networking adapter, processor, etc. In addition to the standard HPCC installation, we included the HPCC plugin for Python. This plugin allows for HPCC to use python code along side ECL. We used Python version 2.7.15, which is installed on each node. Specifically, the use of Python through the pyembed plugin allows us to use virtually any Python library, most importantly *TensorFlow*, see Section 2.8, *Keras*, *NumPy*, and *pandas* [1, 23, 4, 32, 54]. As with Python, each of these Python libraries, and their dependencies, are installed on each node in the clusters.

3.3 IMPLEMENTATION

The coding contribution of this paper is twofold. First are the contributions written in Python and second are the ECL contributions that follow a parallelism paradigm, in this case, *data parallelism*, see Section 2.3. When HPCC distributes the data to a worker node, that data is then processed by the embedded Python code. However, initially the data is formatted for HPCC systems in an ECL Record, detailed in Section 2.7. In order for the Python code to process it, there needs to be a set of functions that interprets between the ECL Record and the Python embed and vice versa. Similarly, since HPCC handles the communication between the nodes, the neural network model itself needs to be interpreted between the pyembed and ECL. Additionally, any function parameters and any meta-data that needs to be passed between nodes also needs to be handled by a pyembed-ECL interpreter. We created, along with the ECL counterparts, various functions to handle these interpretations.

3.3.1 Python

At the beginning of the training processes a neural network needs to be created. We use Python libraries, Keras, and TensorFlow to define our neural network architecture, create our loss functions, and to train the models. Our implementation is designed to be flexible and extensible and can support any architecture supported by either TensorFlow or Keras. This includes neural networks with various types of dropout layers, fully connected layers, convolutional layers that would be used to design a neural network architecture. Once a neural network architecture is decided upon, it needs to be represented in code along with the desired optimizer and loss functions. Defining an MLP [59, 60, 76] in Keras is shown below. It is important to note the difference between defining a model in Keras and in TensorFlow, see Section 2.8, with the former being considerably less lines of code and consequently, quicker.

```
model = Sequential()
model.add(Dense(100, activation='relu', input_shape=(123,)))
model.add(Dropout(0.5))
model.add(Dense(2, activation='sigmoid'))
model.compile(loss=losses.binary_crossentropy,optimizer=Adam(epsilon=1
    e-08),metrics=['accuracy'])
```

Code Example 3.1: Defining a Neural Network with Keras

Once the neural network has been defined, it needs to be trained. Following a Parallel SGD approach, the initialized model needs to be distributed to each worker node along with its partition of training data. The pyembed code, that runs on each node, accepts as parameters the training data[2], model, and meta-data. As the

[2]The first iteration receives the training data, and every iteration after that uses the locally cached partition of data

training starts, the model will be the randomly initialized model and every iteration after this it is the partially trained model up until that point. The first interpreter enables the pyembed code to receive the training data. The training data is stored locally on each node, after being distributed by HPCC, in an ECL Record format with at least one attribute that corresponds to the Y value and some arbitrary number of values that corresponds to the X value, or the features which to learn from. Recall that the Y value is the variable, or category, that the model is trying to learn, see Section 2.1. For example, if the model is being trained on images of hand written digits, the Y value would be the numeric digit that is visually represented by the image and the X value would be the image in some encoded format, in the case of MNIST (hand written digits database [37]) it is the individual grey scale pixel values represented as an integer between 0-255. The interpreter then takes the data passed in by the HPCC Python plugin and converts it into a usable format for training using Keras functions. The final step is to then locally cache this data to disk using the highly optimize HDF5 format [69]. Using this specialized file format to cache data between epochs drastically reduced increased performance between epochs.

```
1 model = Sequential.from_config(...) #pass model configuration from ECL
2 model.compile(loss=losses.binary_crossentropy,optimizer=Adam(epsilon=1
      e-08),metrics=['accuracy'])
3 model.set_weights(...) #sets model weights in preparation of training
4 model.fit(x_dat, y_dat, batch_size=32, epochs=1, shuffle=True, verbose
      =0, validation_split=0.2)
```

Code Example 3.2: Worker Training

Example 3.2: On each worker, the model needs to be defined identically to other workers, then the weights need to be set. The weights will be either the initialized

weights, or weights of the partially trained model. In the case of Parallel SGD, all workers start with the same weights. The weights are then returned after training as shown in the last line.

Once the data is in a format consumable by Keras, the model training can start. In Keras, the process of training the neural network is very flexible. Various optimizer settings, such as batch size, number of epochs, logging, among others, are passed into the Keras function by the meta-data handler. The implementation of this handler was designed to be flexible and extensible to be able to handle a wide variety of model types and training scenarios. The Keras function then starts the TensorFlow training on the TensorFlow model created by the Keras API. After the training process, Keras is used to get the model weight updates from the newly trained model. The workers return the updated model weights in the same format as it received it, so it can run in a recursive fashion.

```
1 updatedWeights = model.get_weights()
2 return (...) #returns to ECL an object with updatedWeights
```

Code Example 3.3: Returning New Weights

Example 3.3 shows how the weights are simply returned from the pyembed to ECL for aggregation.

At this point, all nodes have completed training and the weights need to be aggregated. The way the weights is to be aggregated is dependent on which parallelization scheme the user defines. The scheme can have the model recursively trained for a number of more epochs, or its performance can be evaluated on unseen test data. To properly evaluate a model, it must be used to make predictions on unseen test data. Unseen test data is simply data that has not been directly used in training the model.

A performance metric, depending on what the nature of the mode, is used to show how well the neural network can generalize. The evaluation simply predicts on the features in the test data and compares against the ground truth of that data point. Consequently, since this is essentially using the model, at this point the model can easily be saved to disk for later deployment and consumption by HPCC Systems.

3.3.2 ECL

Our implementation uses ECL code to partition and distribute the training data to each node, the distribution of the pyembed code, the execution of the pyembed code on each node, and the communication of the model between the nodes. As discussed previously, there needs to be an ECL-Pyembed handler to bridge the gap between the two different language. Thus far, we have only discussed the Python side of the implementation.

We employ the data parallel paradigm for distributed neural network training. The entire dataset is partitioned into smaller sets and is distributed to each node. The number of partitions is dependent on two factors, an individual node's memory and the number of nodes in the system. The dataset is first divided by the number of nodes so that each of the N nodes gets 1/N of the data. If this 1/N of the whole dataset can fit into memory, only N partitions are created. If the underlying hardware results in a memory constraint, further partitions are created on each node. For example, we have a 10 node cluster, dataset with 100,000 rows (or X,Y pairs, see Section 2.1), and each node can only fit 5,000 rows in memory, there will be 20 data partitions where each node gets 2 of the partitions. This partitioning is done dynamically and provides flexibility when choosing datasets, training tasks, and hardware. Each partition, or partitions, is then communicated from the master node to the slave nodes, adhering to data parallelism.

Along with the data partitions, ECL transmits the pyembed code to each node

for processing. Once the pyembed code processes its partition, or multiple partitions if memory is limited, the neural network updates are aggregated from the workers to the master node, in the form of an ECL record. Example ECL code is detailed below. It is important to note that the following code is written only once, and HPCC distributes the code as needed to any necessary Thor nodes.

```
mnist_data_type := RECORD
 INTEGER1 label;
 DATA784 image;
END;
```

Code Example 3.4: HPCC Record Definition

```
trainingData := (DATASET('~mnist::train',mnist_data_type, THOR);
dTrainingData := DISTRIBUTE(trainingData);

Marked := PROJECT(dTrainingData, TRANSFORM(nodeMarked, SELF.node :=
    Std.system.Thorlib.Node()+1, SELF:=LEFT), LOCAL);
GroupedData := GROUP(Marked, node, LOCAL);

subMarked := PROJECT(GroupedData, TRANSFORM(smallGroup, SELF.grp:=
    COUNTER DIV M, SELF:=LEFT), LOCAL);
subGroup := GROUP(subMarked, grp, LOCAL);
```

Code Example 3.5: Data and Code Distribution

```
weightUpdates := ROLLUP(subGroup, GROUP, runPy(model, ROWS(LEFT), LEFT
    .grp, 0, FALSE));
```

Code Example 3.6: Distributed Parallel Processing

Example 3.4 denotes an ECL record type definition. A record of this type would have some number of rows, for each image in the database, each with two attributes. A one byte integer representing the image class, and a block of hexadecimals 784 bytes long, representing each of the 784 grey-scale pixels in the 28x28 MNIST image.

Example 3.5 illustrates how ECL can be used to distribute and partition the data and pyembed code from the Thor master node to the Thor slave nodes. The first lines define an ECL dataset from the MNIST file and distribute it across the nodes. Remember that ECL is a declarative language, not an imperative language, so the actual distribution of the data might not happen at this particular point. The next group of code partitions the dataset into N partitions, where N is the number of nodes. The last group of code partitions the data into groups of M, where M is chosen by the programmer to avoid memory constraints.

Example 3.6 takes the actions denoted in Sections 3.4 and 3.5, and perform the distribution and aggregation as outlined by the data parallelism paradigm. The variable *weightUpdates*, would then be further processed according to the parallel optimization scheme.

3.4 PERFORMANCE METRICS

The fundamental reason for training neural networks in parallel is to speed up the the process. Thus, for our implementation to be valid, the training time needs to decrease as the computational resources are increased. Furthermore, the implementation and additional resources should not significantly degrade the performance of the neural

network being trained, i.e. adding nodes to the cluster should train a similarly performant model in less time as compared to one trained on a cluster with fewer nodes, including one trained on a single node system. To validate our implementation we run several case studies on several different clusters sizes and compare the resulting neural networks performance and training time. The models performance metrics and the statistical analysis of the training time are described as follows.

We use two performance metrics to validate our implementation, training time (measured in seconds) and Area Under the receiver operating characteristic Curve (AUC) [9, 63]. The performance of the model, independent from training time, is to used to measure if the cluster size has any significant effects on the training process and the resulting model. Additionally, we split our datasets in specific ways to minimize bias in our models that would arise in which data points are used for training, validation, and testing. We use the performance metrics of a model trained on a single node as our baseline to compare those trained on the different size clusters.

Training neural networks in a parallel fashion tries to reduce the time required to train a model by adding additional computational resource to the problem. However, depending on the training scenario, adding too many nodes can increase training time, due to increased computational overhead. Recall that penalization time complexity is dependent on two factors, one of which is the communication cost between nodes, see Section 2.2. Simply, too many nodes can create a communication cost that outweights the reduction in computation time. We use time, in seconds, as our primary performance metric to evaluate how efficient our parallelization implementations are, how training time is affected by the number of nodes in the system with respect to data size, as well as how data size itself effects the training time. Specifically, we use time to find at which point (when adding nodes to the system) we see diminishing returns in performance and even drops in performance, on a specific hardware setup. In our tests, the shorter amount of time would in general be better.

As we show in Chapter 4, adding nodes does decrease training time as we expect. However, we need to ensure that the reduction in training time does not have a significant reduction in model performance. We use the performance metrics to show there is acceptable or no significant loss in performance due to the parallelization. This dataset used in this binary classification problem is highly imbalanced, as discussed in Section 3.5. AUC shows the trade off between true positive rate (TPR) and false positive rate (FPR) across the range of class distribution and error costs. Simply, AUC provides a numeric representation of how well a model classifier performs in a variety of situations [52], including our imbalanced dataset that has a high number of majority class instances and a small number of minority class instances.

3.5 MEDICARE PART B DATASET

The dataset used in our case studies is a Medicare Part B fraud dataset[3]. It consists of Medicare Part B claims and is used for finding fraudulent physicians. Medicare Part B is a U.S. government program that was created to cover some costs associated with medical procedures, primarily for people age 65 years and older. The dataset is a labeled dataset containing Medicare claims each identified as a fraudulent Medicare claim (by the physician) or not. In general, the claims process is as follows. A physician will perform one or more medical procedures on a patient and the physician then submits a claim to Medicare, rather than directly to the patient, for payment. Additional information can be found in [70, 16] and [71]. The original data was released by the Centers for Medicare and Medicaid Services [15] as a result of a new policy by the U.S. Department of Health and Human Services [28] in an attempt to identify Medicare fraud, waste, and abuse. This original 2015 Part B data is then prepared for use in this thesis' case studies [6]. It consists of over 3.3 million records each with 29 attributes that are then one-hot encoded for use in our neural

[3]We use "MedicareB" as shorthand for the Medicare Part B Fraud Dataset.

networks. There are 15 attributes that correlate to the physician themselves, such as ID number, name, gender, state, and what type of physician they are. In addition, there are 14 more attributes that correlate to the claims procedure, such as where the procedure was performed, medical procedure codes, and different statistical metrics for the Medicare payments. These include, average of the charges that the provider submitted for the service, standard deviation of the Medicare payment amount (the difference from the average cost of this procedure), and others [5]. Each record is roughly 850 bytes long in memory, making this a 2.8 gigabyte file. Described in detail in Chapter 4, this dataset is then divided into several different size datasets for experimental validation. For example, one of the derived datasets has a 1:1 class ratio imbalance. This equates to roughly 2,240 records with a total of roughly 2 megabytes in memory.

3.5.1 Random Under-Sampling (RUS)

The datasets used in this thesis' case studies are derived from the MedicareB dataset, see Section 3.5. We created 10 different size datasets by performing random under-sampling (RUS) on the entire MedicareB dataset. In this dataset, there are roughly 1,400 instances belonging to the minority class, and the rest belong to the majority class. During RUS, a random selection of the majority class is combined with the entire majority class to create a new dataset with a specified class imbalance ratio. For example, if the desired class ratio is 1:1, all of the minority class is combined with a random 1,400 instances from the majority class, to produce a dataset total of 2,800 instances (1,400 minority and 1,400 majority). Importantly, during RUS, the distribution of all other attributes is maintained between the original dataset and the newly RUS created, with respect to the majority class. The minority class is not modified during this procedure. The random under-sampling technique has been shown to improve model classification performance, as compared to training on the

entire dataset without RUS in the domain of Tweet sentiment analysis [53, 72].

CHAPTER 4

CASE STUDIES

This chapter provides two case studies completed to evaluate our implementation's parallel neural network training on HPCC Systems. The methodology of the implementation and the dataset used is discussed in Chapter 3. The methodology for all case studies is the same and the neural network architecture is the same. However, we did use various different data set sizes on different cluster sizes to study their impact on the implementation to validate our work. We conducted 20 experiments on 10 different MedicareB dataset sizes for each of the 5 different cluster sizes for a total of 1,000 observations. The first case study aims to validate the implementation by studying the effects of data size on model training. The second case study details the effects of the number of slave nodes in a cluster, with respect to training a neural network. Finally, we discuss the implementation from a practical standpoint.

4.1 TRAINING TIME SCALABILITY

In this case study we seek to observe how the training time scales with respect to data size and cluster size. We conduct 50 different experiments of varying data sizes on different sized clusters each with 20 trials each for statistical analysis. For each experiment, and on each cluster size, we train an identical MLP model on increasing data set sizes and compare the training time of each. As the dataset size increases, we expect to have the training time increase. Additionally, we expect to see an inverse relationship between training time and cluster size.

In addition to any limitations of the individual algorithms, systems, or frame-

works, the combination poses the possibility of introducing additional limitations. One such limitation is with memory size. Since the training computations are being executed by a Python interpreter, controlled by HPCC, and the code itself is run through the pyembed, there exists additional memory constraints when training a model as compared to training outside of the HPCC environment. This was considered in the implementation, and a setting is provided to the user that would be dependent on the underlying hardware, neural network being trained, and the data, to overcome the systems shortcomings. Additionally, this adds to the robustness of the implementation since it can easily be adapted to different hardware, different data sizes, and different neural network architectures. However, due to these limitations, we expect our implementation to slow with increases in data sizes up until a node's partition of data is too large for the pyembed memory allocation.

We use the MedicareB dataset with Random Undersampling (RUS), see Section 3.5.1, to generate our different dataset sizes. First, the whole dataset is divided into two partitions with 80 percent for training and 20 percent for testing. RUS is performed on the training partition to generate the training datasets of varying sizes. Each experiment uses the same testing dataset size with an even class ratio (e.g. 1:1 ratio). A RUS ratio is used and is in the form of Minority Class to Majority class. Each generated dataset uses the same number of instances from the minority class (1,120 instances). For example, the dataset 1:25 would have 1,120 minority instances and 28,000 majority instances (1,120 * 25 = 28,000) for a total dataset size of 29,120. Our generated datasets used in this study range from 1:1 (2,240 total instances) to 1:2,000 (2,241,120 total instances).

First, we look at Figure 4.1. It can be observed that both the training data size and number of nodes have a large impact on the training time. For a given cluster size, the training time increases with data size. Inversely, for a given dataset size, training time decreases as cluster size increases. Our implementation allows the user to reduce the

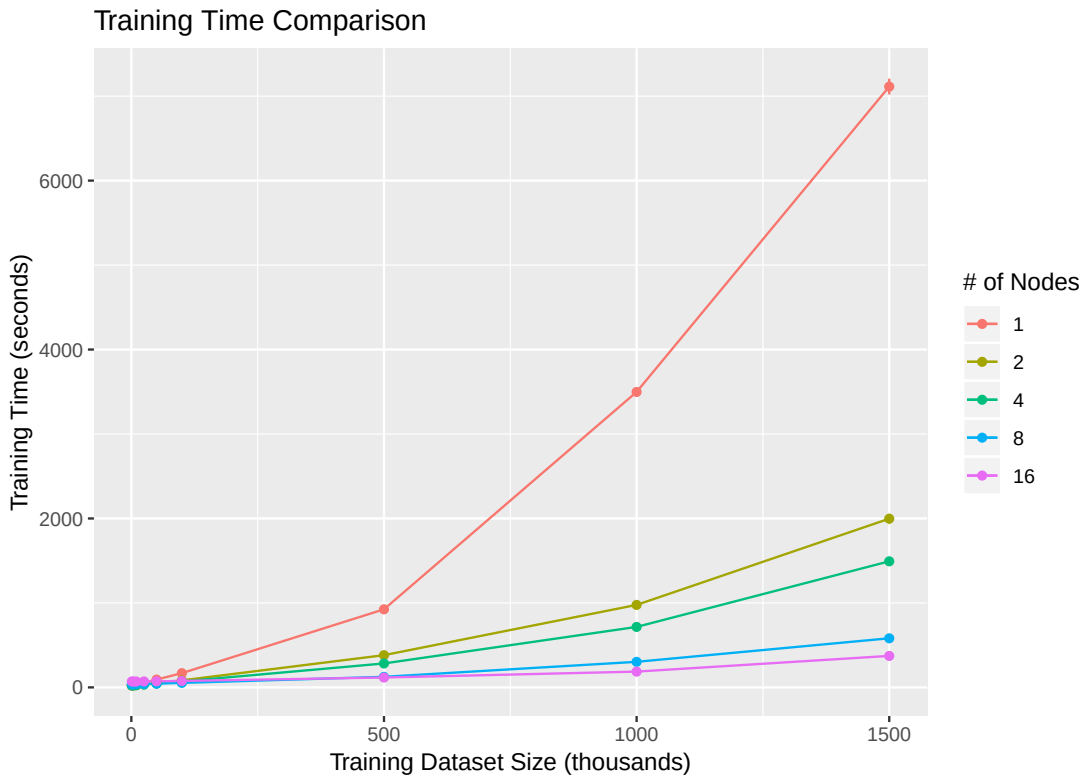


Figure 4.1: Visualization of training time with respect to training data size in thousands. Each line represents results from a cluster with N number of nodes. Note that the X and Y scale are linear.

necessary training time of a model by simply adding nodes to the system. However, it does follow Amdahl's Law[1] [3] in that there is diminishing returns in training time as nodes are added to the cluster. So the benefits of additional nodes would only be realized with sufficient increases in data size. Specifically, a larger cluster should not be slower than a smaller one with enough data.

Consider the scenario where there is either a small training data size or it is not sufficiently large for a certain number of nodes. Additional nodes will in fact increase training time by a constant. Figure 4.2 illustrates this. Looking at the smallest dataset size, clusters with 8 and 16 nodes are orders of magnitude slower than the other cluster sizes. Interestingly, their training time is constant for several of the

[1] Amdahl's Law is a formula that defines the maximum theoretical speedup in executing of a fixed task with increases in performance of the computer.

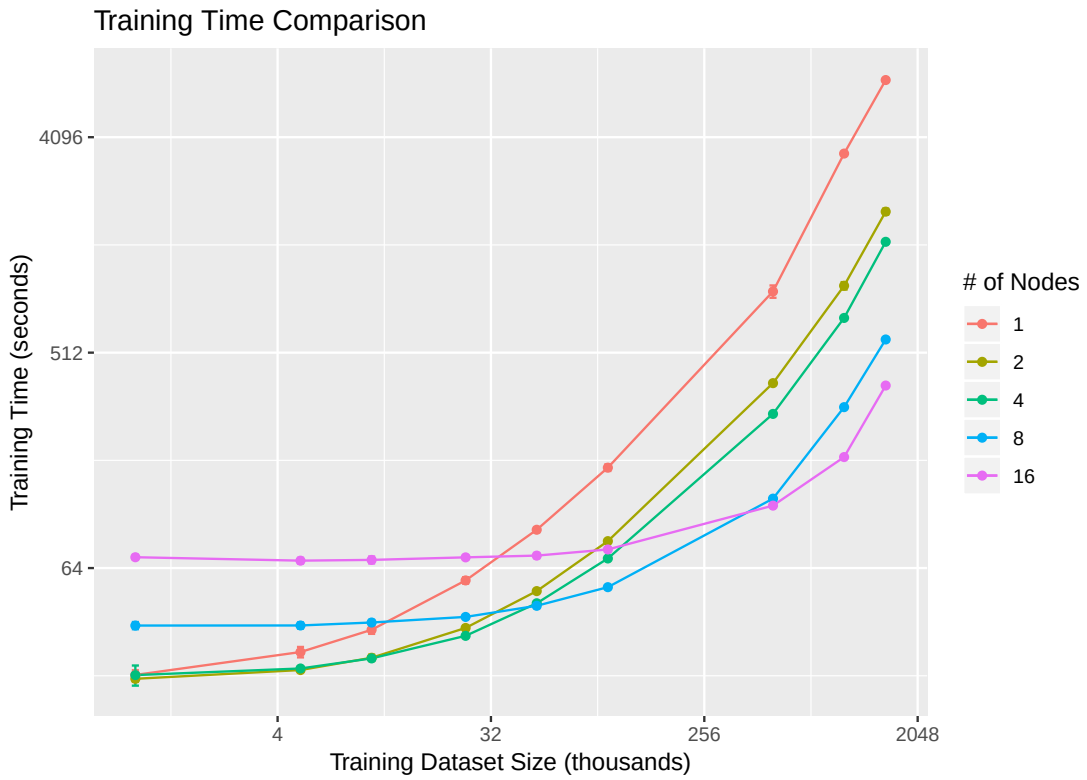


Figure 4.2: Visualization of training time with respect to training data size in thousands. Each line represents results from a cluster with N number of nodes. Note that the X and Y scale are logarithmic, and clearly shows the constant overhead of the larger clusters on small data sets.

smaller data sizes. The dataset size that is large enough to increase training time, for a given node count, is the same size at which we start to see a benefit of using a cluster of that size. This very clearly shows the situation when the constant communication cost outweighs the benefits of the increased processing power, see Section 2.2. It is important to note the X scale of the graph is logarithmic.

It is interesting to note the difference between the training time vs. cluster size curve between the datasets with 1:2,000 and 1:1,500 class ratio. Figure 4.2 illustrates that there is very little, if any, difference between the two training time curves. Comparing any other two sequential datasets (i.e. any two datasets that are closest in size) there is a significant change in training time when increase the training size. This is due to the relationship between the physical hardware specifications and the

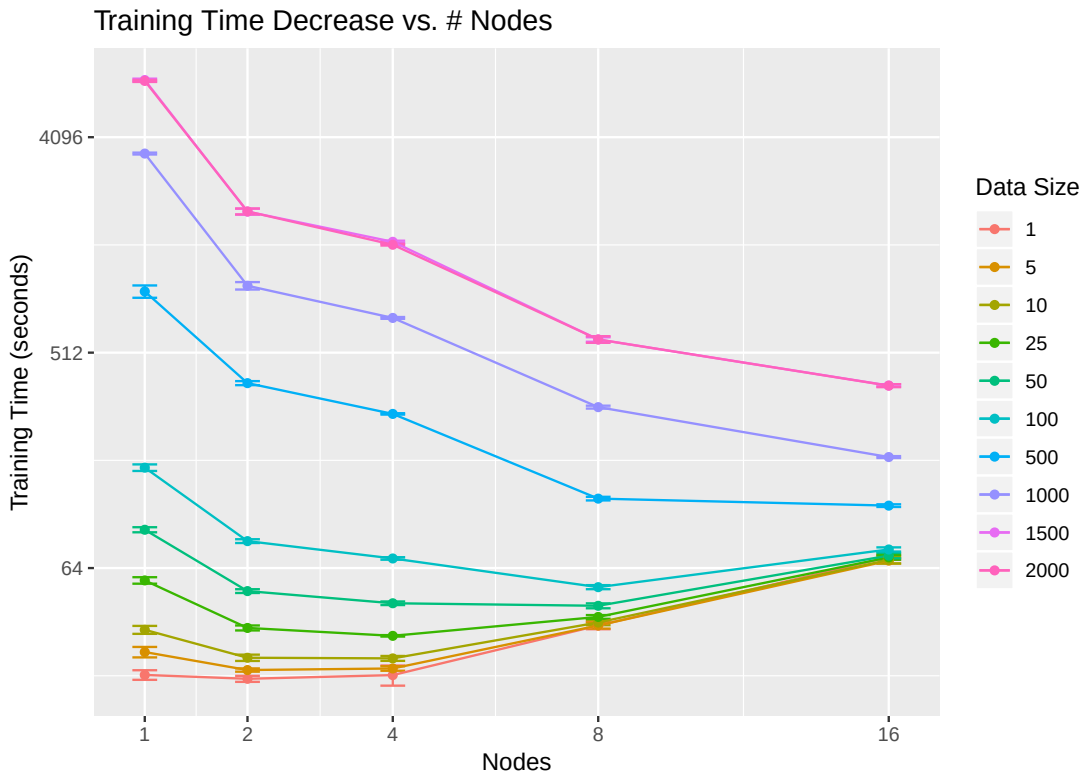


Figure 4.3: Visualization of training time with respect to cluster size. Each line represents results from the different dataset sizes used in in the case study. Note the smaller training sets increase training time with additional nodes and the largest datasets decrease training time with increased nodes.

size of the dataset on disk in this thesis. Recall that the implementation allows for larger than memory datasets to be trained. To accomplish this, the dataset is partitioned into smaller segments and sequentially trained on by a given node. The small percentage difference between the 1:1,500 dataset and the 1:2,000, as compared to the percentage differences between other pairs, combined with the fact that on every experiment, those two datasets ended up being on the same size partitions are why their training time curves are so similar. Additionally, the increased communication time, with increased cluster size, can be observed by the upward sloping line segments correlating to the smaller data set sizes, see Figure 4.2. Only after a certain size will the time curve continually slope downward, in this figure, the dataset with 1:500 class imbalance.

4.1.1 Statistical Analysis

We grouped the experiments into three distinct groups. The first group presents a dataset with a 1:1 class ratio. This group shows how the implementation performs on a small dataset. The second group presents the datasets with 1:5 and 1:10 class ratios. These two datasets are grouped since they are the smallest datasets in which the addition of worker nodes proves beneficial. The final group presents the datasets with 1:25, 1:50, 1:100, 1:500, 1:1,000, 1:1,500, and 1:2,000 class ratios. These are grouped because each successive dataset has the same trend of reduce communication costs and increased utility of additional nodes as the previous. The trends are presented in a series of box plots in Figures 4.6 and 4.7. We use Analysis of Variance (ANOVA) [7] and Tukeys Honestly Significant Difference (HSD) tests [2] to determine if dataset size and number of training nodes in the cluster significantly impact model training time [63]. All of the experiments are included in the ANOVA tables and HSD tables. Only the first analysis subsection, for the 1:1 class ratio dataset, includes the two different tables in line with the text. For the remainder of this thesis, all remaining ANOVA and HSD tables are found in the Appendix on page 61. All references are in the format of A.#.

Dataset 1:1 Class Ratio Analysis

First, we examine the case where the dataset has a 1:1 class ratio, the smallest size used in our thesis. This dataset as 1,120 majority records and 1,120 minority records, hence 1:1 ratio. We discussed previously that a dataset needs to be sufficiently large to benefit from more nodes and can even run slower if there are too many parallel nodes. In our case study, clusters with 1, 2, and 4 worker nodes all have statistically similar training times for the dataset 1:1. We plot the results of training time vs node count in a box plot format (such as in Figure 4.4). Each box depicts the 20 trials for each experiment. Outliers are denoted by single points, and the box itself

denotes the mean, median, and 25^{th} and 75^{th} percentiles. The box plot is testing for the significance of the number of nodes in a cluster with respect to training time of a given model trained on a given dataset. Boxes that are similar in Y values denote no significant difference in training time between those two cluster sizes. Boxes that have different Y values, have significant difference in training time between those two cluster sizes. The results are illustrated here, and are numerically analyzed below and are presented in Appendix .

As shown in Figure 4.4, the cluster with 8 nodes is statistically significantly slower than the smaller clusters, and the 16 node cluster is slower still. The results are tested with an ANOVA and are presented in Table 4.1 showing that the number of nodes has a significant effect on training time. In addition to the ANOVA values, we present the Tukey's Honestly Significant Difference test (HSD)[2] to compare and rank the factors. Tables 4.1 and 4.2 show that our assumptions from the visualizations in Figures 4.4 and 4.2 are valid.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Nodes	4	35419.46	8854.87	3968.24	0.0000
Residuals	95	211.99	2.23		

Table 4.1: ANOVA Table showing the number of nodes in a cluster is a statistically significant factor regarding training time training time on the 1:1 dataset size.

	TrainingTime	groups
16	71.06	a
8	36.74	b
1	22.82	c
4	22.78	c
2	21.98	c

Table 4.2: Tukey HSD test with groupings. Training time vs. # nodes, 1:1. The slowest is the 16 node cluster, second slowest is the 8 node cluster, and all clusters in group "C" have no difference in training time.

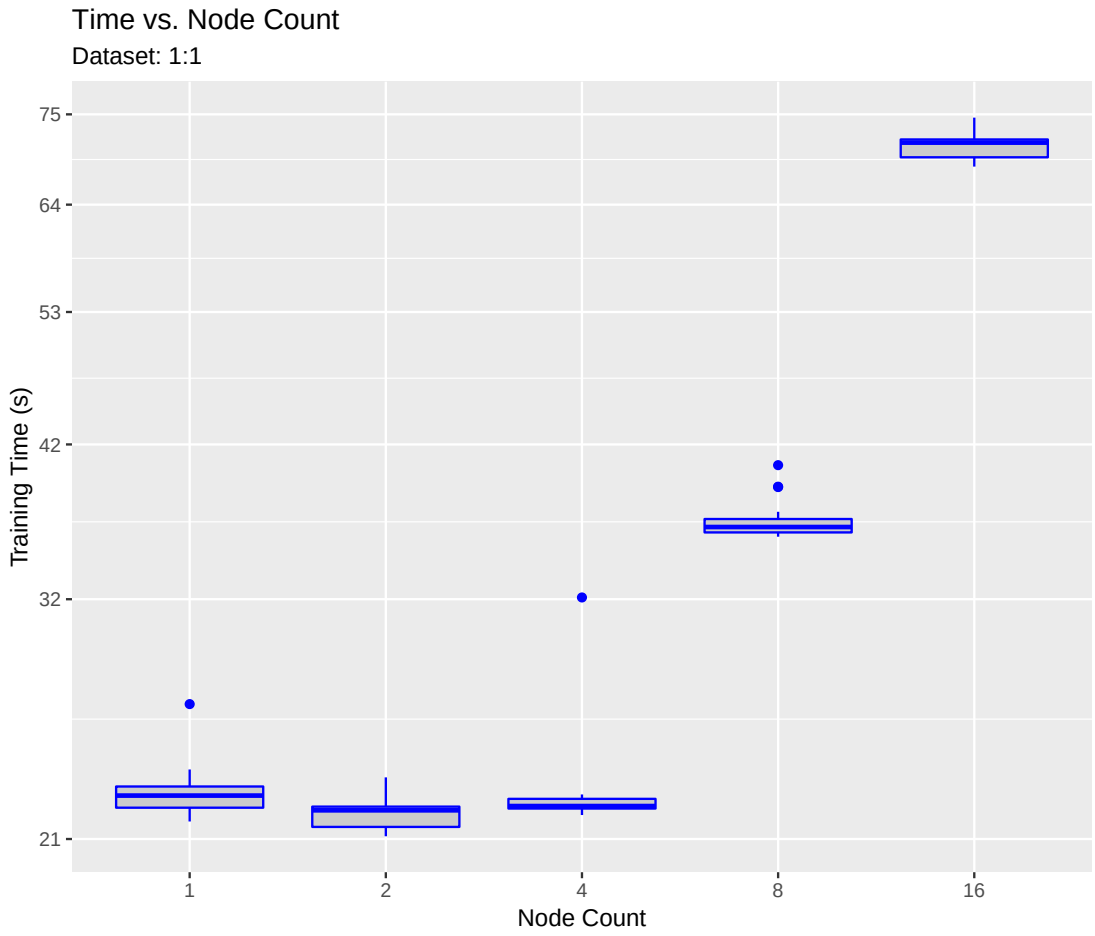
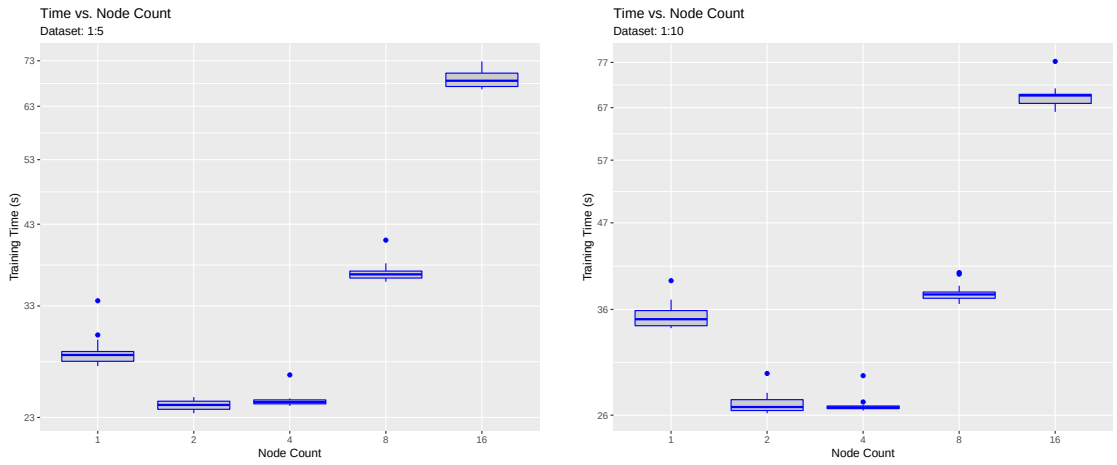


Figure 4.4: depicts a box plot comparing the training time vs. node count for the dataset size 1:1.



(a) 1:5 (b) 1:10

Figure 4.5: depicts a box plot comparing the training time vs. node count for the dataset sizes 1:5 and 1:10.

Datasets: 1:5 & 1:10 Class Ratio Analysis

Next, we examine the case where the dataset size is 1:5 and 1:10, the next smallest data sizes. Recall that the 1:5 dataset is 6,720 records long and the 1:10 is 12,320 records long, both relatively small training set sizes. However, the results show that these are the first sufficiently large datasets to benefit from an increase in nodes. Figure 4.1.1 shows a drop in training time when comparing a 1 node cluster to a 2 node cluster, for the 1:5 dataset (a) and the 1:10 (b) dataset respectively. Similar to the previous dataset size, 1:1, the clusters with larger numbers of nodes are showing significant increase in training time due to the communication costs. Tables 1 (1:5) and 3 (1:10) again show the number of nodes is a significant factor in training time. Tables 2 and 4 show a single node cluster or clusters with 16 or 8 nodes take significantly longer to train on both the 1:5 dataset and the 1:10 dataset. In this case, the 4 node and 2 node clusters perform similarly, for each dataset. Thus, training times would be sped up by adding nodes, with similar results, for both datasets.

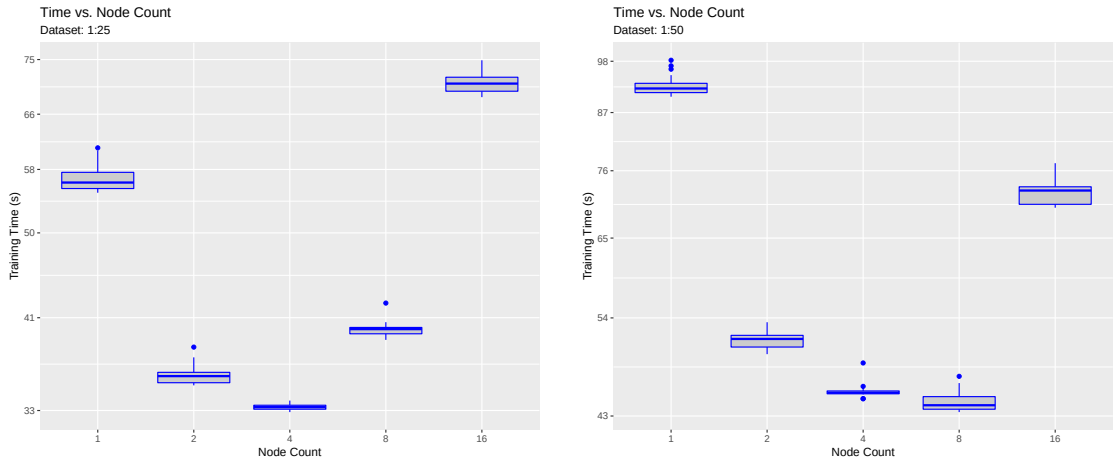
Datasets: 1:25, 1:50, 1:100, 1:500, 1:1,000, 1:1,500, & 1:2,000 Class Ratio Analysis

Next, we examine the rest of the datasets (1:25, 50, 100, 500, 1,000, 1,500, 2,000), ranging from a size of 29,120 records to a size of 2,241,120 records. The previous trend of increasing utility per additional node as the data set grows, continues with these larger datasets. Figures 4.6(a) and 4.6(b) for datasets 1:25 and 1:50 show that the larger clusters sizes' utility are steadily increasing, though still negative (i.e. larger clusters are slower). Referring to Table 8 and figure 4.6(b), a dataset of at least 57,120 is needed before an 8-node system is worth considering. Though training time for an 8-node and a 4-node cluster are grouped together, indicating they train in the same amount of time, any larger dataset than this would train significantly faster with a cluster with at least 8 nodes, see Table 10 and 12. Finally, it can be observed from Table 14 and Figure 4.7(a) a training data set of 1,121,120, or greater is sufficiently large to warrant the use of a cluster with at least 16 nodes.

4.2 MODEL PERFORMANCE SCALABILITY

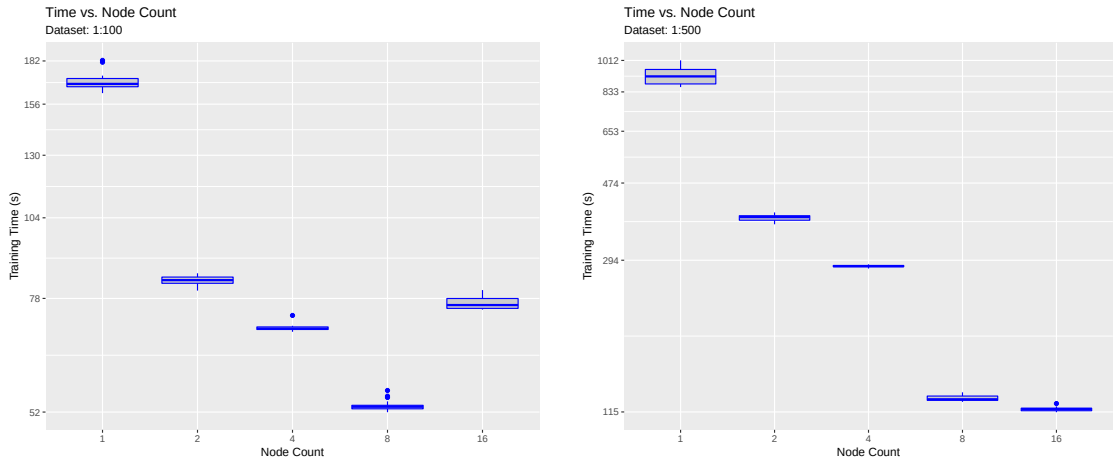
In this case study, we seek to observe our implementation's impact on model performance. We conduct the same experiments on the same cluster sizes as in the previous case study, see Section 4.1; however, we use AUC as our model's performance metric. Similar to the previous case study, we trained the same MLP model. It is a binary classification model that is used for fraud detection on the MedicareB dataset, see Section 3.5. Though higher AUC indicates better model performance, we focus on relative change of AUC. Specifically, the change in AUC observed while varying cluster size.

First a base line AUC was generated by training different models on different data sizes on a cluster with a single node and is used to compare against the results obtained on the other clusters sizes. Clusters with 1, 2, 4, 8, and 16 nodes are evaluated. Typically larger data sets produce better performing deep learning models.



(a) 1:25

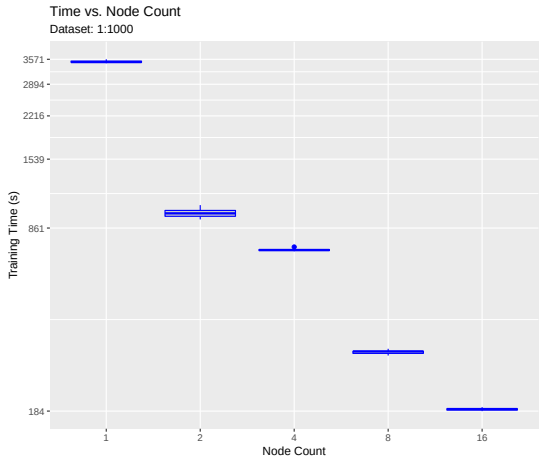
(b) 1:50



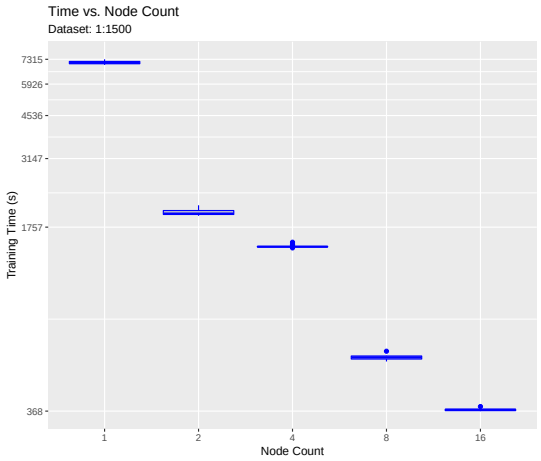
(c) 1:100

(d) 1:500

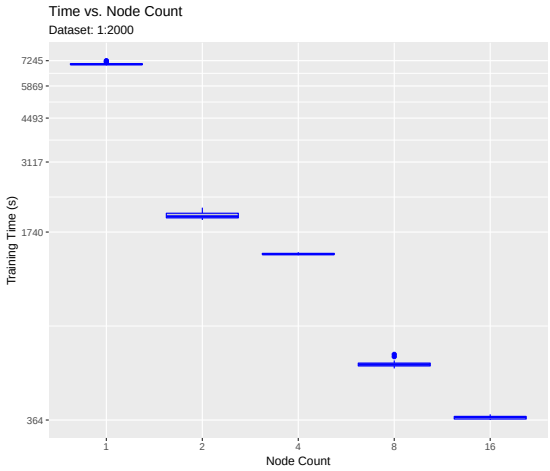
Figure 4.6: depicts a box plot comparing the training time vs. node count for the dataset sizes 1:25, 1:50, 1:100 and 1:500. Note the increasing utility of extra nodes as the dataset increases.



(a) 1:1,000



(b) 1:1,500



(c) 1:2,000

Figure 4.7: depicts a box plot comparing the training time vs. node count for the dataset sizes 1:1,000, 1:1,500 and 1:2,000. Note the dataset sizes are sufficiently large to warrant a cluster of at least 16 nodes.

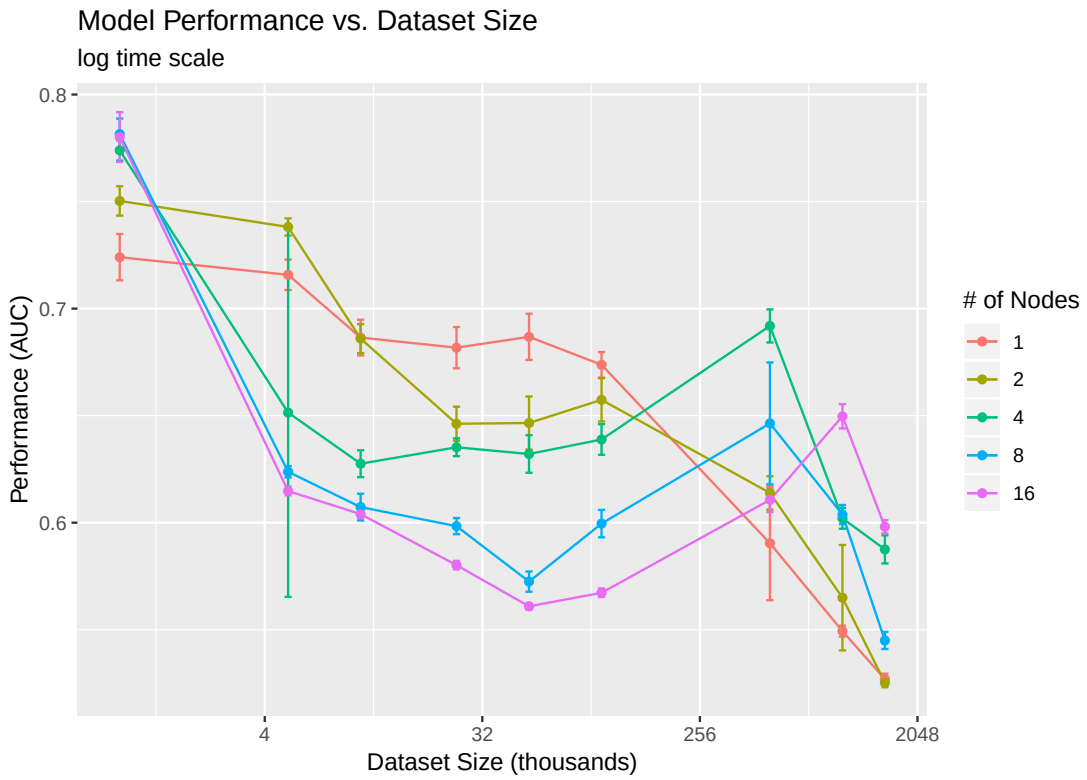


Figure 4.8: Line graph illustrating the effects of dataset size and cluster size on model performance. Note the general negative trend in performance with respect to both cluster and dataset size.

However, the data sets used in this case study have varying levels of class imbalance. Class imbalance has been shown to present unique challenges and negative effects to model performance [33, 62, 72]. It is important to note that the difference in performance between different dataset sizes, though different, is out of scope of this case study. It can be observed from Figure 4.10 that the AUC trends lower as the dataset size increases. While this may seem counterintuitive since a neural network is expected to perform better with additional data, recall that, other than the 1:1 dataset, each dataset is an imbalanced dataset. Each dataset used was created using random under-sampling (RUS) from the original MedicareB dataset. For example, the dataset 1:500, where a significant drop in performance can be observed in Figure 4.10, is a 500 to 1 majority to minority class imbalance. There are roughly 500,000 instances of a majority class and roughly only 1,000 instances of a minority class.

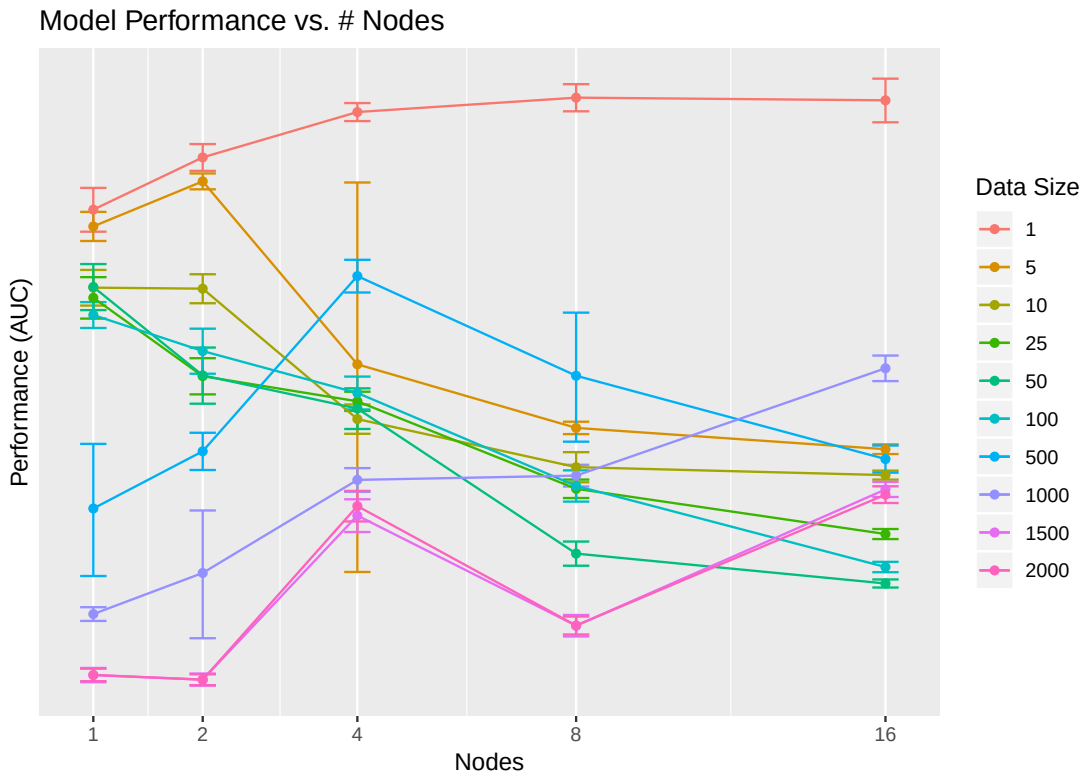


Figure 4.9: Line graph illustrating the effects of dataset size and cluster size on model performance. Note the general negative trend in performance with respect to both cluster and dataset size.

This severe class imbalance is likely the cause of the model performance degradation with increased data size [33, 72, 62].

4.2.1 Statistical Analysis

The significance of the results presented in this case study were tested by performing ANOVA with 5% confidence level. In addition to the ANOVA, we present Tukey's Honestly Significant Difference test to compare the factors. The ANOVA table presented in Table 19 shows the number of nodes in a cluster is a significant factor with regards to model performance. The HSD Table 20 presents the increased model performance with increased number of nodes. In this table, the 8 and 16 node clusters perform similarly. Figure 4.2.1(e) visualizes the results from the prior two tables.

Table 21 presents the ANOVA results for the dataset 1:500 and Table 22 presents

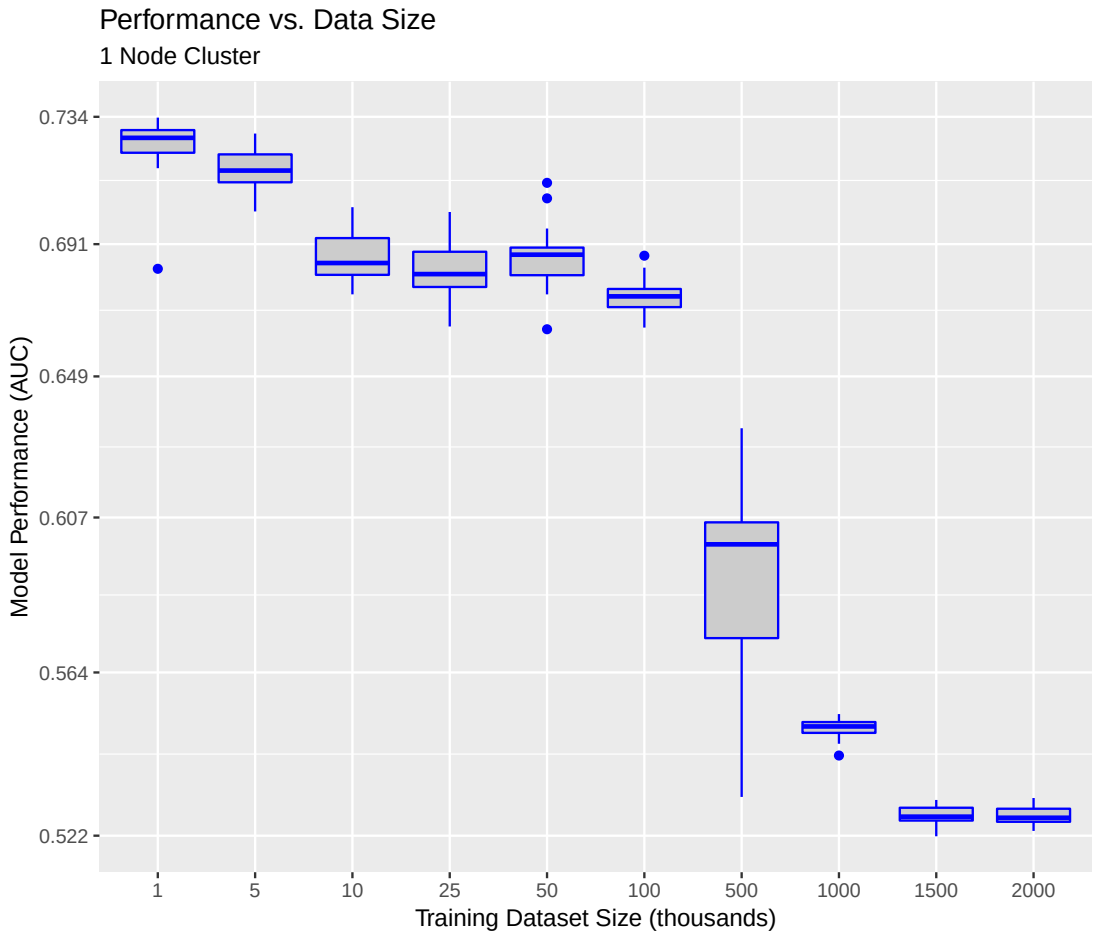
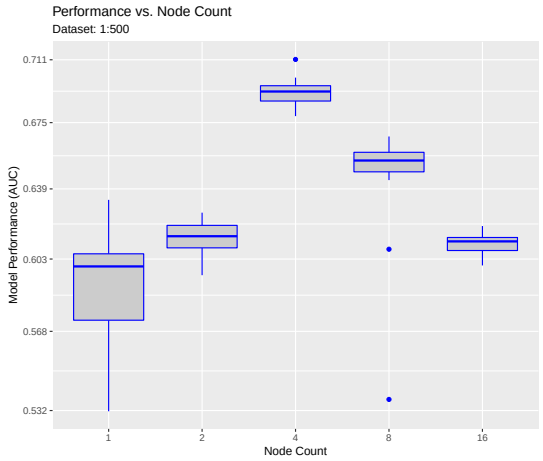
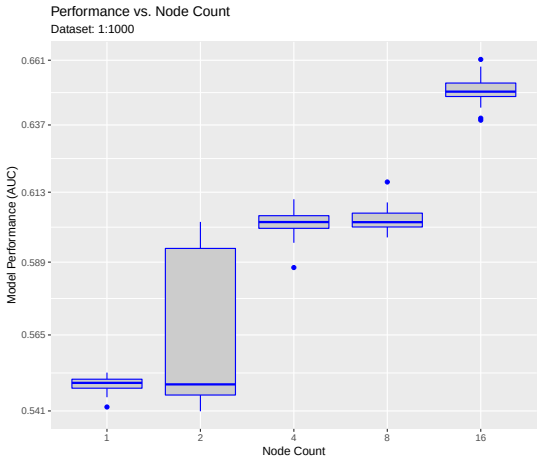


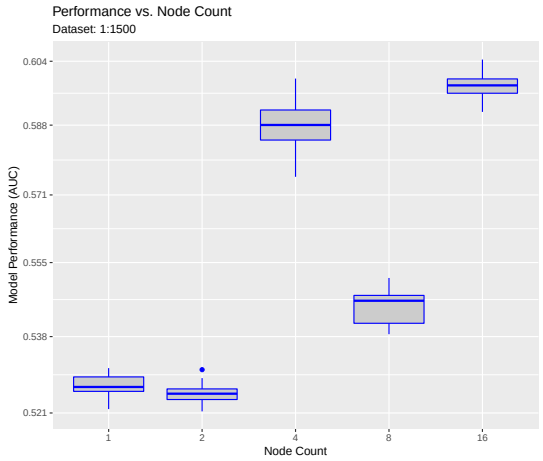
Figure 4.10: Visualization of model performance with respect to cluster size. Note the increased drop in performance with the dataset size of 500.



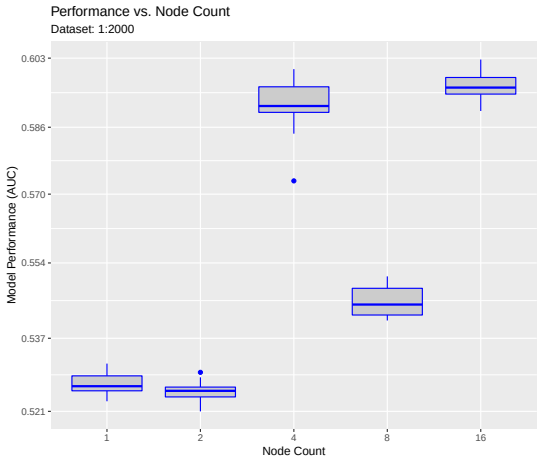
(a) 1:500



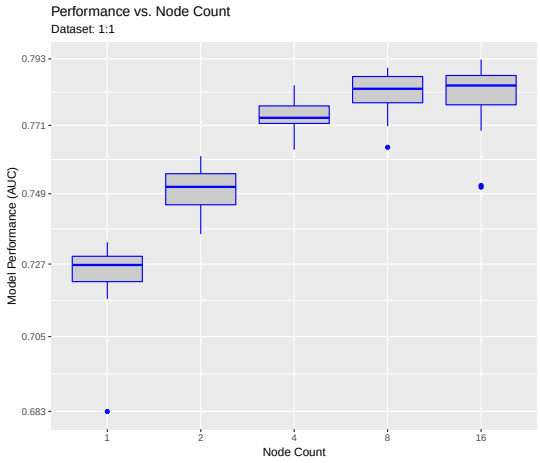
(b) 1:1,000



(c) 1:1,500



(d) 1:2,000



(e) 1:1

Figure 4.11: Box plots of model performance vs node count for different dataset sizes. Note the increased performance for the 1:1, balanced, dataset (e).

the HSD rankings of the clusters' model performance. Figure 4.2.1(a) visualizes the results of the ANOVA and HSD test. It can be observed that model performance increases up until 4 nodes, and decreases with subsequent larger clusters.

Table 23 presents the ANOVA results for the dataset 1:1,000 and Table 24 presents the HSD rankings of the clusters' model performance. Figure 4.2.1(b) visualizes the results of the ANOVA and HSD test. The results from the 1:1,000 dataset closely resembles that of the 1:1 dataset in that both their model's performance is continually benefited from increasing cluster sizes. Additionally, this dataset is sufficiently large enough that in addition to the added model performance, the training time is also reduced with larger clusters sizes.

Table 25 presents the ANOVA results for the dataset 1:1,500 and Table 26 presents the HSD rankings of the clusters' model performance. Figure 4.2.1(c) visualizes the results of the ANOVA and HSD test. This shows the effects on model performance is rather erratic with clusters of size 4 and 16 performing better than the rest, with 16 being the highest performer. This combined with the added benefit of the reduced training time, this specific dataset would benefit the most from the 16 node system.

Table 27 presents the ANOVA results for the dataset 1:2,000 and Table 28 presents the HSD rankings of the clusters' model performance. Figure 4.2.1(d) visualizes the results of the ANOVA and HSD test and are very similar to those of the 1:1,500 dataset, with similar conclusions.

4.3 DISCUSSION

As seen in the results of our case studies, the required time to train a neural network in a distributed and parallel fashion is greatly impacted by both the dataset size as well as the number of worker nodes in the cluster used. However, each additional worker added to a cluster increases the communication cost of training. Communication cost is independent of the dataset size. It is dependent on the underlying network speed,

size of the neural network, and the total number of worker nodes. For a larger system to train faster than a smaller one, the individual node's additional computational power needs to offset the communication cost. Thus, a sufficiently large dataset is required to realize any training time gains. Our case study proves that the minimum size to be considered "sufficient" can be quite small. For a two-node cluster, a dataset size of only 6 kilobytes (the 1:5 class imbalance is 6,720 rows long, each row 850 bytes, for roughly 6 kilobytes in size) is sufficiently large to see a reduction in training time by using a 2-node cluster versus a single node cluster. Since the speed up is also dependent on the underlying hardware specifications, a user would need to consider this factor when choosing the cluster size to use. As discussed in the second case study, the resulting model performance is affected by the cluster size. For a small balanced dataset, and some larger unbalanced datasets, the addition of a few nodes can increase model performance. Since the implementation's general trend shows an increase in model performance with an increase in nodes, and thus faster model convergence, this would allow for fewer training epochs to achieve the same desired model performance. Additionally, it has been shown that increasing the number of nodes decreases training time. Both of these combine to further reduce training time, for a desired model outcome, making this distributed deep learning implementation ideal.

CHAPTER 5
CONCLUSION AND FUTURE WORK

Training state of the art neural networks requires very large data sets and large amounts of computing power. As the dataset sizes increase and deep neural networks grow in complexity, so too does the computational and memory demands. One approach to solve this is to train a model across multiple computers in a distributed and parallel fashion. By dividing up the required computations across a cluster, the overall training time can be reduced. We implemented Parallel Stochastic Gradient Descent on HPCC Systems to achieve this. We combined the popular neural network training library TensorFlow with the powerful distributed cluster system, HPCC, to achieve a synchronous data parallel method for training neural networks. In this chapter, we provide conclusions for our research and parallel deep learning implementation and provide future work that will compliment this thesis.

5.1 CONCLUSIONS

In our first case study, we examined how our proposed implementation reduces the required training time for a neural network with respect to dataset size and cluster size. We found the time required in a parallel computation is dependent on two factors. The increase in the available computational power by adding to the parallelism will reduce the training time but the additional communication, which grows with each additional degree of parallelism, will add to the overall computation time. Thus, the training time is highly dependent on both the data size and cluster size. The communication cost is a constant function of the neural network architecture size and the

number of parallel workers in the cluster. The time to train a model on a dataset is generally dependent on the size of the data, bigger data equates to longer training times. In our case study, we presented the balance between cluster size and dataset size to minimize the training time using MedicareB data. Generally, only larger data is able to effectively take advantage of large numbers of nodes due to communication overhead. For example, our results indicated a minimum dataset size of 1,000,000 records is needed for a cluster with 16 worker nodes. With small datasets, such as the 2,000-record long dataset, there was no significant difference in training time between clusters with 1, 2 and 4 nodes.

Since training time decreases are not very productive if they produce neural networks with significantly worse classification performance, our second case study explored model classification performance. Our research was limited to the MedicareB dataset without any class imbalance. We trained models on the 1:1 class imbalance dataset and found that there exists an improvement in model performance when training on larger clusters. Specifically, the training time between 1, 2, and 4 node clusters were the same; thus time would not be a factor in choosing which cluster size to use. However, since the model performance was significantly improved using the larger cluster, it would be optimal to train on the 4-node cluster.

From these results, we conclude that the distributed parallel neural network implementation on HPCC Systems is effective at reducing training time without introducing negative effects on the classification performance of a trained model for MedicareB data. Additionally, the implementation is effective across a wide range of dataset sizes and is capable of scaling across several cluster sizes. To achieve optimal training time reductions, the size of the cluster must be decided with the specific dataset size at hand as well as the underlying system hardware.

5.2 FUTURE WORK

The implementation presented in this thesis provides the basis for future research and development of distributed parallel neural network training on HPCC Systems. Our implementation is limited to being synchronous and is a data parallel method. Additionally, our case studies presented validation using one type of neural network architecture and data from one domain. Potential avenues for future work to expand upon this implementation or to enhance the research done in the case studies is included below.

- Design and implementation of a model parallel method of distributed training for HPCC Systems. These methods have an advantage when the model architectures become huge. Furthermore, a combination of these two, a hybrid model and data parallel approach, would benefit HPCC Systems by allowing it to train both very large models on very large data sets on commodity hardware.

- Implementation of additional parallel paradigms on HPCC Systems would be worthwhile specifically, implementation of various asynchronous optimizers. These methods present new complexities such as stale parameters and an increased communication overhead.

- Additional research validating it across more domains and more model architectures would complement this work.

APPENDIX

ANOVA AND HSD TABLES

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Nodes	4	28247.64	7061.91	4840.01	0.0000
Residuals	95	138.61	1.46		

Table 1: Training Time - Number of Nodes 1:5 ANOVA

	TrainingTime	groups
16	68.75	a
8	36.77	b
1	28.46	c
4	24.28	d
2	23.92	d

Table 2: Tukey HSD test with groupings. Training time vs. # nodes, 1:5

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Nodes	4	24510.20	6127.55	3322.10	0.0000
Residuals	95	175.23	1.84		

Table 3: Training Time Number of Nodes 1:10 ANOVA

	TrainingTime	groups
16	69.25	a
8	37.84	b
1	35.27	c
2	26.93	d
4	26.76	d

Table 4: Tukey HSD test with groupings. Training time vs. # nodes, 1:10

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Nodes	4	20631.91	5157.98	3681.09	0.0000
Residuals	95	133.12	1.40		

Table 5: Training Time - Number of Nodes 1:10 ANOVA

	TrainingTime	groups
16	70.98	a
1	56.81	b
8	39.94	c
2	35.91	d
4	33.28	e

Table 6: Tukey HSD test with groupings. Training time vs. # nodes, 1:25

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Nodes	4	34671.41	8667.85	3891.79	0.0000
Residuals	95	211.59	2.23		

Table 7: Training Time - Number of Nodes 1:10 ANOVA

	TrainingTime	groups
1	92.66	a
16	72.18	b
2	51.25	c
4	45.57	d
8	44.47	d

Table 8: Tukey HSD test with groupings. Training time vs. # nodes, 1:50

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Nodes	4	163360.77	40840.19	5822.21	0.0000
Residuals	95	666.38	7.01		

Table 9: Training Time - Number of Nodes 1:10 ANOVA

	TrainingTime	groups
1	168.72	a
2	83.02	b
16	76.60	c
4	70.23	d
8	53.23	e

Table 10: Tukey HSD test with groupings. Training time vs. # nodes, 1:100

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Nodes	4	8775639.96	2193909.99	3561.74	0.0000
Residuals	95	58516.80	615.97		

Table 11: Training Time - Number of Nodes 1:10 ANOVA

	TrainingTime	groups
1	924.28	a
2	381.50	b
4	283.44	c
8	125.12	d
16	116.94	d

Table 12: Tukey HSD test with groupings. Training time vs. # nodes, 1:500

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Nodes	4	147538415.96	36884603.99	81140.16	0.0000
Residuals	95	43185.00	454.58		

Table 13: Training Time - Number of Nodes 1:10 ANOVA

	TrainingTime	groups
1	3498.11	a
2	976.49	b
4	715.89	c
8	302.60	d
16	186.87	e

Table 14: Tukey HSD test with groupings. Training time vs. # nodes, 1:1000

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Nodes	4	595817698.86	148954424.72	56721.27	0.0000
Residuals	90	236346.94	2626.08		

Table 15: Training Time - Number of Nodes 1:10 ANOVA

	TrainingTime	groups
1	7113.30	a
2	1997.29	b
4	1492.72	c
8	581.39	d
16	372.73	e

Table 16: Tukey HSD test with groupings. Training time vs. # nodes, 1:1500

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Nodes	4	592359783.04	148089945.76	82896.55	0.0000
Residuals	92	164352.73	1786.44		

Table 17: Training Time - Number of Nodes 1:10 ANOVA

	TrainingTime	groups
1	7052.96	a
2	2000.89	b
4	1450.84	c
8	581.50	d
16	372.25	e

Table 18: Tukey HSD test with groupings. Training time vs. # nodes, 1:2000

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Nodes	4	0.05	0.01	162.40	0.0000
Residuals	95	0.01	0.00		

Table 19: Performance - Number of Nodes 1:1 ANOVA

	ModelPerformance	groups
8	0.78	a
16	0.78	ab
4	0.77	b
2	0.75	c
1	0.72	d

Table 20: Tukey HSD test with groupings. Model Performance # nodes, 1:1

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Nodes	4	0.13	0.03	94.35	0.0000
Residuals	95	0.03	0.00		

Table 21: Performance - Number of Nodes 1:500 ANOVA

	ModelPerformance	groups
4	0.69	a
8	0.65	b
2	0.61	c
16	0.61	c
1	0.59	d

Table 22: Tukey HSD test with groupings. Model Performance # nodes, 1:500

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Nodes	4	0.12	0.03	221.44	0.0000
Residuals	95	0.01	0.00		

Table 23: Performance - Number of Nodes 1:1000 ANOVA

	ModelPerformance	groups
16	0.65	a
8	0.60	b
4	0.60	b
2	0.56	c
1	0.55	d

Table 24: Tukey HSD test with groupings. Model Performance # nodes, 1:1000

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Nodes	4	0.09	0.02	1451.49	0.0000
Residuals	90	0.00	0.00		

Table 25: Performance - Number of Nodes 1:1500 ANOVA

	ModelPerformance	groups
16	0.60	a
4	0.59	b
8	0.54	c
1	0.53	d
2	0.53	d

Table 26: Tukey HSD test with groupings. Model Performance # nodes, 1:1500

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Nodes	4	0.09	0.02	1689.06	0.0000
Residuals	92	0.00	0.00		

Table 27: Performance - Number of Nodes 1:2000 ANOVA

	ModelPerformance	groups
16	0.60	a
4	0.59	b
8	0.55	c
1	0.53	d
2	0.53	d

Table 28: Tukey HSD test with groupings. Model Performance # nodes, 1:2000

BIBLIOGRAPHY

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] H. Abdi and L. J. Williams. Tukeys honestly significant difference (hsd) test. *Encyclopedia of Research Design. Thousand Oaks, CA: Sage*, pages 1–5, 2010.

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[4] Apache Software Foundation. Keras.io. https://www.ubuntu.com/.

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

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

[7] M. L. Berenson, M. Goldstein, and D. Levine. *Intermediate Statistical Methods and Applications: A Computer Package Approach 2nd Edition*. Prentice Hall, 1983.

[8] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[9] A. P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997.

[10] R. E. Bryant. Data intensive scalable computing. *Carnegie Mellon University. Retrieved August*, 10:2009, 2008.

[11] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[13] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

[14] D. Ciregan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649, June 2012.

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

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

[17] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with cots hpc systems. In *International Conference on Machine Learning*, pages 1337–1345, 2013.

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

[19] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[20] J. Dean and S. Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[21] A. Frome, G. S. Corrado, J. Shlens, S. Bengio, J. Dean, T. Mikolov, et al. Devise: A deep visual-semantic embedding model. In *Advances in neural information processing systems*, pages 2121–2129, 2013.

[22] GitHub.com. State of the octoverse, 2017.

[23] Google. Tensorflow. https://www.tensorflow.org/.

[24] I. Gorton, P. Greenfield, A. Szalay, and R. Williams. Data-intensive computing in the 21st century. *Computer*, 41(4):30–32, 2008.

[25] A. Hadoop. Apache hadoop, 2011. Software available from hadoop.apache.org.

[26] G. Heigold, V. Vanhoucke, A. Senior, P. Nguyen, M. Ranzato, M. Devin, and J. Dean. Multilingual acoustic models using distributed deep neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8619–8623. IEEE, 2013.

[27] J. Herath, Y. Yamaguchi, N. Saito, and T. Yuba. Dataflow computing models, languages, and machines for intelligence computations. *IEEE Transactions on Software Engineering*, 14(12):1805–1828, 1988.

[28] HHS. U.s. department of health human services. http://www.hhs.gov/.

[29] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[30] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, volume 1, page 3, 2017.

[31] W. E. Johnston. High-speed wide area, data intensive computing: a ten year retrospective. 1998.

[32] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–2018. Online; accessed 8/1/18.

[33] T. M. Khoshgoftaar, C. Seiffert, J. Van Hulse, A. Napolitano, and A. Folleco. Learning with limited minority class data. In *Machine Learning and Applications, 2007. ICMLA 2007. Sixth International Conference on*, pages 348–353. IEEE, 2007.

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

[35] Q. V. Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013.

[36] 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*, pages 265–272. Omnipress, 2011.

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

[38] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[39] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[40] C. J. Maddison, A. Huang, I. Sutskever, and D. Silver. Move evaluation in go using deep convolutional neural networks. *arXiv preprint arXiv:1412.6564*, 2014.

[41] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[42] Microsoft. Azure instance types. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-general.

[43] Microsoft. How to create and deploy a cloud service, 1975-2018. https://docs.microsoft.com/en-us/azure/cloud-services/cloud-services-how-to-create-deploy-portal.

[44] Microsoft. Truly consistent hybrid cloud with microsoft azure. 2017.

[45] A. M. Middleton and A. Chala. Hpcc systems: Introduction to hpcc (high-performance computing cluster). *White paper, LexisNexis Risk Solutions*, 2011.

[46] N. Murata, S. Yoshizawa, and S.-i. Amari. Network information criterion-determining the number of hidden units for an artificial neural network model. *IEEE Transactions on Neural Networks*, 5(6):865–872, 1994.

[47] M. M. Najafabadi, T. M. Khoshgoftaar, F. Villanustre, and J. Holt. Large-scale distributed l-bfgs. *Journal of Big Data*, 4(1):22, 2017.

[48] M. M. Najafabadi, F. Villanustre, T. M. Khoshgoftaar, N. Seliya, R. Wald, and E. Muharemagic. Deep learning applications and challenges in big data analytics. *Journal of Big Data*, 2(1):1, 2015.

[49] T. Nordstrom and B. Svensson. Using and designing massively parallel computers for artificial neural networks. *Journal of parallel and distributed computing*, 14(3):260–285, 1992.

[50] OpenAI. OpenAI Five. https://blog.openai.com/openai-five/.

[51] M. Pethick, M. Liddle, P. Werstein, and Z. Huang. Parallelization of a back-propagation neural network on a cluster computer. In *International conference on parallel and distributed computing and systems (PDCS 2003)*, 2003.

[52] F. J. Provost, T. Fawcett, R. Kohavi, et al. The case against accuracy estimation for comparing induction algorithms. In *ICML*, volume 98, pages 445–453, 1998.

[53] J. Prusa, T. M. Khoshgoftaar, D. J. Dittman, and A. Napolitano. Using random undersampling to alleviate class imbalance on tweet sentiment data. In *2015 IEEE International Conference on Information Reuse and Integration*, pages 197–202, Aug 2015.

[54] pydata. Pandas — python data analysis library. https://pandas.pydata.org/.

[55] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[56] D. Reinsel, C. Chute, W. Schlichting, J. McArthur, S. Minton, I. Xheneti, A. Toncheva, and A. Manfrediz. The expanding digital universe. *White paper, IDC*, 2007.

[57] M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE, 1993.

[58] J. E. Rodrigues and J. E. Rodriguez Bezos. A graph model for parallel computations. 1969.

[59] F. Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[60] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[61] D. Saad. Online algorithms and stochastic approximations. *Online Learning*, 5, 1998.

[62] C. Seiffert, T. M. Khoshgoftaar, J. Van Hulse, and A. Napolitano. Mining data with rare events: a case study. In *ictai*, pages 132–139. IEEE, 2007.

[63] N. Seliya, T. M. Khoshgoftaar, and J. Van Hulse. A study on the relationships of classifier performance metrics. In *Tools with Artificial Intelligence, 2009. ICTAI'09. 21st International Conference on*, pages 59–66. IEEE, 2009.

[64] N. B. Serbedzija. Simulating artificial neural networks on parallel architectures. *Computer*, 29(3):56–63, 1996.

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

[66] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[67] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

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

[69] The HDF Group. Hierarchical Data Format, version 5, 1997-2018. http://www.hdfgroup.org/HDF5/.

[70] U.S. Government, U.S. Centers for Medicare Medicaid Services. The offical u.s. governement site for medicare. https://www.medicare.gov.

[71] U.S. Government, U.S. Centers for Medicare Medicaid Services. Whats medicare? https://www.medicare.gov/sign-up-change-plans/decide-how-to-get-medicare/whats-medicare/what-is-medicare.html.

[72] J. Van Hulse, T. M. Khoshgoftaar, and A. Napolitano. Experimental perspectives on learning from imbalanced data. In *Proceedings of the 24th international conference on Machine learning*, pages 935–942. ACM, 2007.

[73] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008.

[74] J. Varia and S. Mathew. Overview of amazon web services. *Amazon Web Services*, 2014.

[75] T. White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[76] L. Yann. *Modeles connexionnistes de lapprentissage.* PhD thesis, PhD thesis, These de Doctorat, Universite Paris 6, 1987.

[77] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, et al. On rectified linear units for speech processing. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 3517–3521. IEEE, 2013.

[78] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

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

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

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