Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Engineering

Design and Implementation of a Distributed Neural Network Platform Utilising Crowdsource Processing

Simon Fitzgerald

12309625

Supervisor: Stefan Weber

A dissertation submitted in partial fulfilment of the degree of

M.A.I. (Computer Engineering)

Submitted to the University of Dublin, Trinity College, May 2018

Declaration

I, Simon Fitzgerald, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

_____ _____

Signature Date

Acknowledgements

I would like to thank my supervisor Stefan Weber for providing me with the opportunity to work on this project, meeting with me throughout the project's duration and offering ideas on how best to proceed with the research, encouragement and understanding throughout the year.

I would also like to thank those who took their time to proof-read this report and provide me with help and feedback during my research; Daniel, Evan, Gwen and Kris.

Summary

Neural networks are at the forefront of future technologies, with self-driving cars on the horizon, and complex classification algorithms controlling what we see in many parts of our daily life already. Neural networks and deep learning thrive on large datasets and large networks, both of which contribute to longer training times that impede model development and deployment in business and research endeavours.

The possibility to make use of these techniques are often out of reach for the majority of applications due to expensive Graphics or Tensor Processing Units needed, coupled with the long computational times. This project aims to alleviate this issue by distributing the training process across multiple willing users, forming an online distributed training network across loosely connected, heterogeneous systems.

This was achieved through the use of the TensorFlow[1] library in tandem with a peer to peer file replication algorithm for data distribution across workers. Machines are initially connected to one another through a central match making server, and otherwise feature no future central architecture interaction. In testing the process was found to improve overall training accuracy by 3-6% per worker over the same number of data iterations, proving this method can allow for network training to converge quicker to the local minimum as compared to a standard solitary machine system.

Table of Contents

Chapter 1.

Introduction

This chapter provides an introduction and high-level description of the background to the project. It also details the objectives of the project, and the anticipated challenges. Finally, it outlines the structure of the remainder of the report.

1.1 Motivation

Machine learning, specifically neural networks, have already become a pervasive part of society today, unknown to many consumers. Recommender systems on popular sites such as YouTube or Amazon[0], targeted advertisement on Facebook or Google[0], stock trading prediction software[2], bank financial fraud prevention[2], and autonomous vehicles, are all just a small part of what neural networks are capable of, and they are set to continue to play even larger roles in the future. Despite their capabilities, due to the computational complexity they are out of reach for many individuals and small organisations which could make use of them.

A major point of motivation for this research was the 'SETI at Home'[3] and 'Folding at Home'[4] projects which are currently ongoing. Both of these projects allow users to donate their computational power to aid in the search for extraterritorial life and disease research through protein folding, respectively. This is implemented via a browser. The client only needs minimal input from the server, which can then be calculated in-browser using the client's local processing and the results returned to the server.

Taking inspiration from this approach of crowd sourced processing, a design was constructed to replicate this model for use with distributed neural network training.

1.2 Objectives

One of the largest issues with neural networks are that they are computationally intensive and correspondingly time expensive to train adequately. Generally speaking, there are two ways to increase processing throughput when it comes to any processing in computing: using higher quality machines or distributing the workload across more machines. This project will focus on the latter and has the overall goal to design and implement a platform which allows for the creation of a heterogenous distributed network, composed of independent machines to train a single neural network as effectively as possible.

From the outset it was obvious it would not be possible to wholly replicate the previous approaches via a lightweight browser-based system, due to the dependent frameworks which are needed for neural networks and the size of the data being used. Thus, an all-in-one application was decided to be the best approach, acting as a single platform for both hosting and working on neural networks, and allowing interaction through a simple interface that any individual would be capable of using.

1.3 Outline

Chapter 2 of this report provides further background information on this project. It explains the software and libraries used as part of this project, along with the necessary fundamental knowledge on the topic of neural networks. Furthermore, it introduces some examples of related research and practical applications.

Chapter 3 details the design of the model, beginning with the problem formulation and moving on to solve each issue in turn. Multiple solutions are proposed, and the chosen solution is defined. Finally, it outlines the overall architecture of the design and breaks it down into stages which need to be implemented.

Chapter 4 explains how each aspect of the aforementioned design was executed in the final model. Each stage of the design is explained, detailing the reasoning and approach taken, as well as any problems which were encountered.

Chapter 5 explores the results obtained throughout the course of this research, along with issues which arose during the evaluation of the final platform, and limitations which prevented full evaluation of the model.

Chapter 6 completes the report with a set of conclusions which have been reached based on the results obtained throughout the project's development. Potential applications of the current model are presented, along with a discussion of the potential future work which could be conducted to improve it.

Chapter 2

Background and Related Work

This chapter begins with an overview of some of the key software components used as part of this research. It then provides information regarding the fundamentals of neural networks which this research builds upon, and a discussion of the current research into the field of distributed neural networks and deep learning, along with the current cutting-edge platforms and services currently available.

2.1 Neural Networks

Artificial Neural Networks (ANN) are the focus of this research. They are popularised by their similarity to biological neurons in the brain, in which a series of neurons 'fire' and stimulate a response in the brain. Unlike their biological counterparts, artificial neurons do not have a binary output, allowing for a range of output values from each neuron. A neuron is given a number of inputs from a dataset, or from other neurons, and outputs a response. Composing a number of these neurons in an interconnected network is what forms an ANN.

2.1.1 Single Layer Perceptron

In its simplest form, a neural network can be expressed as a single layer perceptron shown in Figure 1. A number of inputs are supplied and multiplied by random weight values, these weights are then summed and finally the output is passed through an activation function, σ. Optionally a neuron bias, b, can be added to this sum, resulting in the final output of the neuron represented as the following[7] :

$$\mathbf{output} = a^i = \sigma\left(\sum (w^i \cdot \mathbf{input}^i) + b^i\right)$$

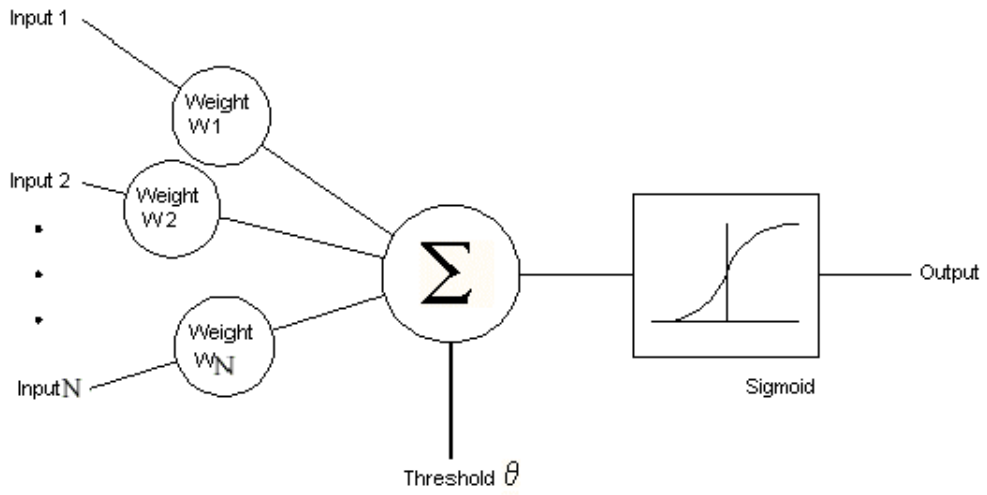


Figure 1: *Single layer perceptron showing multiple weighted inputs, which are summed and passed through a sigmoidal activation function* [6]

As we increase the complexity and expand on this, we get a feedforward multi-perceptron network, shown in Figure(2). This works under the exact same premise as the single layer perceptron, only using intermediary activation functions at each neuron and the outputs are also used as inputs for other neurons. Because of the multiple layers of neurons, updating the weights becomes more intricate, which will be discussed in more detail later.

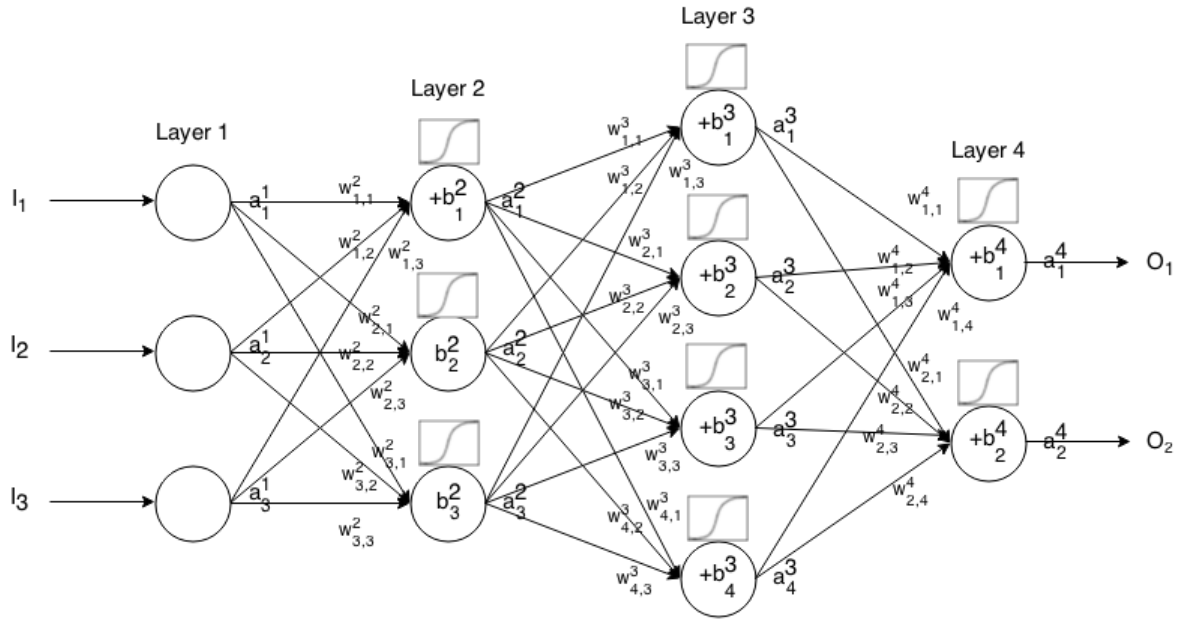


Figure 2*: Multi-layer perceptron neural network, each neuron can be viewed as a single perceptron discussed previously. Combined they form the network shown, which can handle more complex tasks.* [7]

Our formula is updated to reflect this multi-layer set up is given by[7] :

$$a^i = \sigma\left(\sum (w^i \cdot a^{i-1}) + b^i\right)$$

2.1.2 Cost Functions

The final output expressed in the output layer neurons must be evaluated. To do this it is compared to the desired output and a cost function, J(θ), is used to assess the associated cost of the forward pass through the network. The cost function takes in all the weight values of the network, and outputs a single cost value, C.

Quadratic cost, also known as root mean squared cost, function is the simplest to understand and widely used for this reason[7] .

$$C = n^{-1} \sum^n (y^i - \hat{y}^i)^2$$

Where:

Y is the true value of the target

$\hat{y}$ is the estimated value by the network

In this project we are using the cross-entropy cost function, also known as Bernoulli negative log-likelihood and binary cross-entropy[7] , as it was found to give more accurate results in training.

$$C = -n^{-1}\sum^n (y^i \ln a(x^i) + (1 - y^i) \ln(1-a(x^i))$$

Where:

X is the set of input examples

Y is the corresponding target values

2.1.3 Gradient Decent

At this point it is simplest to imagine the current state of the system as a 3-dimensional graph, with the cost function as a plane on this graph and just two weights as the inputs, such as in the example in Figure(3).

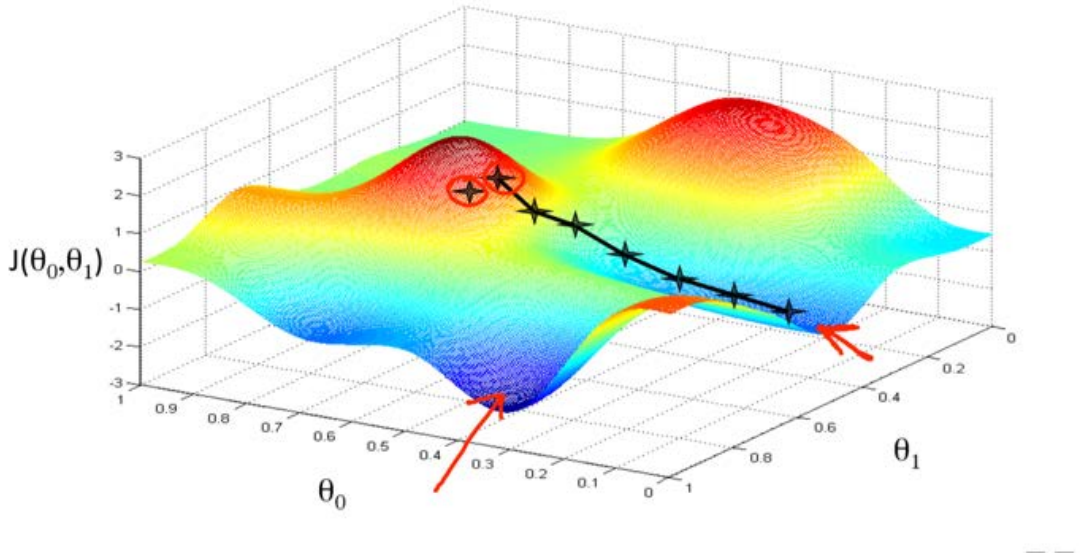


Figure 3: 3D representation of gradient decent [8]

Using the cost output from the previous section a vector for the negative gradient of the function can be calculated which decreases the cost output most quickly. This is then used in the following section - Back Propagation - to nudge the weights of the system, and this process of repeatedly nudging the input of the function by a multiple of the negative gradient is what is called gradient descent. It is a method to converge on some local minimum of a cost function.

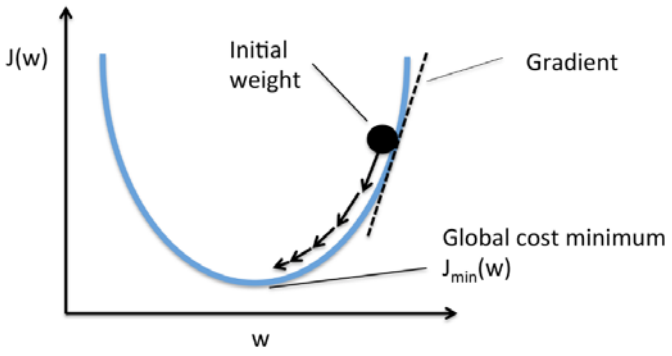


Figure 4: 2D representation of gradient descent [8]

In reality the system is working in far higher dimensions, from the hundreds of thousands to the millions depending on the number of connections in the network, which is very difficult to conceptualise visually.

The rate at which these updates occur can be varied. Using the Gradient Descent (GD) optimization algorithm, often referred to as batch GD, weights are updated incrementally after each pass through the entire data set. This results in only taking a single step closer to converging on the local minimum after each pass through all data, leading to much longer training times with larger datasets. Stochastic Gradient Descent (SGD) updates the weights after each input pass through the network. This causes a 'zig-zag' effect on the path to the local minimum, but updates at a faster rate. The final approach, that will be used in this project, is Mini-Batch Gradient Descent (MB-GD), acting as a midway choice between the previous two approaches. For this the weights are updated after small groups of training samples, allowing MB-GD to converge in fewer iterations than GD but utilizes vectorized operation for computational performance in comparison to SGD[8] .

2.1.4 Back Propagation

Back propagation is the means by which each gradient descent step is calculated[9] .

Let $\mathbf{z^L = w^L + a^{(L-1)} + b^L}$ where **L** is the final layer of the network. The final output of a neuron on this layer is $\mathbf{a^L}$, which is simply the activation function of $\mathbf{z^L}$, $\boldsymbol{\sigma}(\mathbf{z^L})$, similar to what was discussed in the section on a single layer perceptron. This value $\mathbf{a^L}$ is what is compared to the desired value of the output, **y**, and used in conjunction with the cost function to get the cost of that particular output, **C**.

At the heart of back propagation is an expression for the partial derivative, $\mathbf{\partial C/\partial w}$, of the cost function with respect to any weight, or bias, in the network[9] .

For $\mathbf{w^L}$ example,

$$\mathbf{\partial C/\partial w^L = \partial z^L/\partial w^L \cdot \partial a^L/\partial z^L \cdot \partial C/\partial a^L}$$

and thus, giving us the sensitivity of **C** to small changes in a particular weight $\mathbf{w}^L$. Expanding on this using a simple Quadratic Cost Function we get the following set of equations for finding the relative component changes based on the change in cost:

$$\partial C/\partial a^L = 2(a^L - y)$$

$$\partial a^L/\partial z^L = \sigma'(z^L)$$

$$\partial z^L/\partial w^L = a^{(L-1)}$$

As each neuron has multiple connections to it, the sum of cost is used when evaluating. Simply put, back propagation is just applying the chain rule backwards, beginning from the final layer of the network and iterating backwards through the network for all the connections, propagating the original error for each node with respect to the next layer. This is done while feeding inputs forward and adjusting the weights of the neurons to understand how changing specific weights and biases is changing the rate of error[10] .

2.1.5 Hyperparameters & Optimisation

Hyperparameters are network parameters which are set by the network before the training process begins, in contrast with other network parameters which are generated during the training process, such as the neuron weights and biases. Examples of hyperparameters are the learning rate, batch size or number of epochs in the network.

The optimal values for these will drastically vary depending on the network, and greatly impact the overall efficiency of the training process. Hyperparameter optimisation is the process by which the optimum state for these values is achieved[11] .

2.1.6 Neural Network Architectures

Aside from the simple feed-forward multilayer perceptron (MLP) network shown previously, there are many different neural network variations which use varying levels of complexity to perform specific tasks better.

Deep Neural Networks

A deep neural network (DNN) is the same as a simple feedforward network, except that it has multiple hidden layers between the input and output layers. The number of neurons required in a shallow neural network grows exponentially with task complexity and require far more neurons than a deep neural network[13] . Overall the exact source of the efficiency of deep neural network is up for debate, but they have been empirically proven to work better. This efficiency can be seen below in Figure(5) from the *Deep Learning* book from Goodfellow et al 2016. DNNs overall excel in complex neural network tasks.

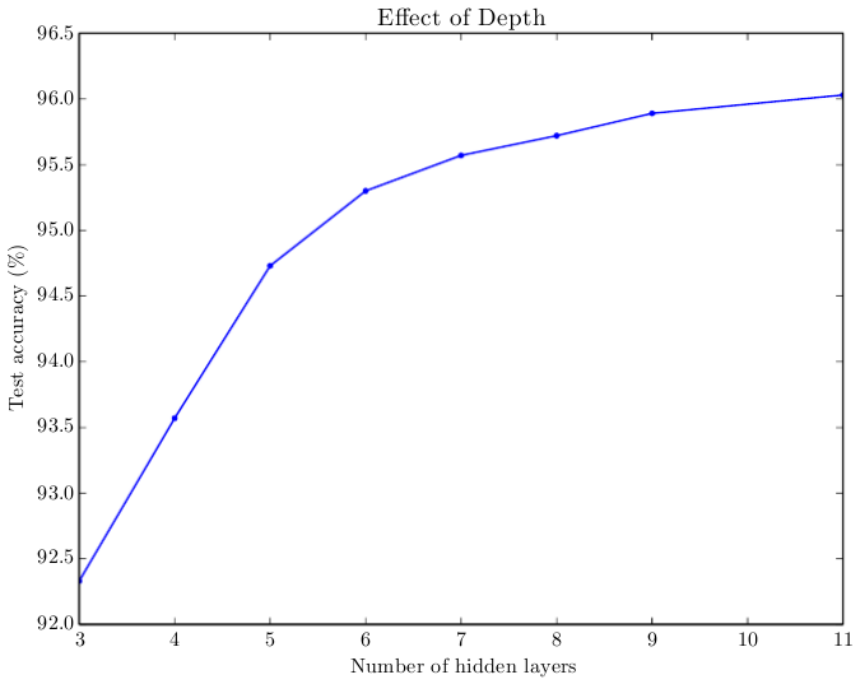


Figure 5*: Data from Goodfellow et al. 2016 Deep Learning book, showing the effect of increasing number of hidden layers in a deep neural network and the resulting network accuracy [13]*

Recurrent Neural Networks

A recurrent neural network (RNN) is a recursive neural network variant in which the neuron connections are directed in a cycle. This means, unlike in feedforward networks, that outputs depend on the previous step's neuron state as well as the present inputs. This memory function allows these forms of networks to excel in problems such as speech or hand writing recognition[12].

Convolutional Neural Networks

A convolutional neural network (CNN) contains one or more convolutional layers that are pooled or fully connected and uses a variation of DNN previously discussed[12]. Each layer uses a convolution operation to the input passing the result to the next layer and allows the network to be deeper with fewer parameters. These forms of networks have been shown to excel in applications relating to computer vision[12].

General Adversarial Neural Networks

General adversarial neural network (GAN) is typically comprised of two DNNs, which are pitted against one another. They work by having one network which generates new instances of data, while the other evaluates data from the generator network and a valid dataset and attempts to verify data authenticity. The former network's goal is to have its data verified by the latter system as authentic data[14]. This results in both networks becoming better at performing their tasks.

2.2 Relevant Software

This section details the frameworks which the platform makes use of use of in this project. It gives some background on the software, its functionality and why it is used in this research.

2.2.1 TensorFlow

TensorFlow[1] is an open source library developed by Google Brain for use in machine learning. TensorFlow is one of the foremost industry standards, being particularly prevalent in the area of neural networking, and it is the main library which will be used for the purpose of this project. TensorFlow is a high performance numerical computation library with flexible architecture, which allows for it to be used to parallelize the training process over local CPUs, GPUs, or Tensor Processing Units (TPU). The tutorials and pre-built functionality from TensorFlow are what lay the foundation for this project.

2.2.2 gRPC Remote Procedure Call

gRPC Remote Procedure Call (gRPC)[5] is a high-performance RPC framework replacement for remote procedure call developed by Google. Its origin goes back to the original single general-purpose RPC infrastructure used by Google named 'Stubby'. It has since evolved into an open-source universal framework used within Google for connecting all Google services and by many other prominent industry leaders such as Cisco and Netflix.

gRPC allows for connecting services in microservice style architecture and connecting devices and clients to backend services. It is highly efficient and allows for bi-directional streaming using the http/2 based transport [5]. This is used within the final platform for communication between the workers and parameter server for transferring updated gradients as quickly as possible in a best effort to overcome network bottlenecking.

2.3 Related Work

This section will cover different research papers which have discussed areas of distributed neural networks, similar to the research that I am pursuing in this project. Each section begins with the title of the research paper upon which it is based, followed by an overview of the work that was done and finally any remarks on the research.

"Distributed learning of CNNs on heterogeneous CPU/GPU architectures"

This paper[18] by Marques, Falcao, Alexandre, 2017 addresses the problems of long training times in larger deep neural networks, specifically CNNs, and aims to alleviate this through distribution of the processing across multiple machines. Due to the inherent properties of CNN layers, they incorporate a unique variation of model parallelism distribution which takes advantage of this natural parallelism in CNNs, and only distributes the convolutional layer of the network, which takes between 60% and 90% of the overall processing time[18].

Their paper reports an achieved increase in speedup of 3.2x when using four CPUs, and 2.45x when using three GPUs[18]. These results were taken using the CIFAR-10 dataset and just two convolutional layers. More modern datasets that require higher complexity, should be expected to cause an increased speedup accordingly.

Overall the paper is a good delve into augmented ways in which neural networks could be distributed outside of standard model and data parallelism.

"Large Scale Distributed Deep Networks"

This paper[19] by Dean et al., 2012 at Google, forms the foundation for the work done in this current project. It is the paper that first applied the parameter-server idea to deep learning, an approach which is still going strong, although other aspects of the architecture such as a lack of distributed GPU support[19] have made the framework overall irrelevant today. DistBelief, the name of the architecture, is also what paved the way and eventually became TensorFlow[1], discussed previously in section 2.1.1.

"Large scale distributed neural network training through online distillation"

This paper[20] by Anil et al., 2018 aimed to increase the rate at which training can be completed. Distillation is the process of training many models in order to achieve a single more refined training model[21]. Typically, it adds increased complexity to the training pipeline, but Anil et al. solve this by distributing the distillation process online over multiple machines[20] .

It is not a dissimilar approach to what is produced in this report, and they claim they were able to fit large datasets twice as fast and that the method is a cost-effective way to make exact predictions of a model more reproducible.

"Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour"

This paper[22] from Goyal et al. 2017 from Facebook makes use of a mini-batch stochastic gradient descent variation, accompanied with a scaling learning rate. The training begins in a 'warm up' stage with a low learning rate to overcome optimisation issues, and gradually increase this. It uses a linear scaling rule, increasing the batch size through the training, and each iteration where the batch is multiplied by a value k, also multiply the learning rate by the same k value[22] .

By following this method and distributing the training in a data parallel model across 256 GPUs, Goyal et al. were able to decrease the training time on the ImageNet dataset from multiple days to just 1 hour[22] , really demonstrating the possibilities of distributed neural networks.

2.4 State of the Art

This section discusses the platforms and software frameworks being used in the world today, how they work, and the functionality they contain.

2.4.1 Distributed File Systems

Large volumes of data are the backbone of neural networks training and evaluation. Managing and accessing this data efficiently is a large issue, which is currently often solved by the use of distributed file systems.

Hadoop

Apache Hadoop[15] is a framework that allows the distribution of large datasets across clusters of computers. It uses a design which allows for scaling from a single server to thousands of machines. The Hadoop Distributed File System (HDFS) provides high throughput access to application data, and Hadoop YARN allows for parallel processing of these large datasets.

YARN (Yet Another Research Manager) is a resource management layer of Hadoop added in Hadoop 2.x. It consists of a Resource Manager, a Node Manager for each slave node in the cluster, and an Application Manager[15]. This system allows for data processing systems to run and process data stored in the HDFS. It is called through the MapReduce component of Hadoop which takes care of scheduling and monitoring tasks.

Spark

Apache Spark[16] is an open-source cluster computing framework. It provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. It claims to offer the potential to achieve workloads of more than 100 times faster than a default Hadoop Distributed File System. Spark runs on Hadoop or Kubernetes, often in combination with Hadoop YARN, replacing the MapReduce component of Hadoop. Spark has been shown to be able to sort 100TB of data 3 times faster than Hadoop MapReduce and is able to effectively handle up to petabytes of data without issue. Spark uses caching of partial results across its memory of distributed workers, in contrast to MapReduce which is completely disk-orientated. MapReduce also uses a batch processing system, while Spark offers real-time processing. Together this results in reduced latency with Spark [16].

PySpark[17] is a python package which allows for interfacing with resilient distributed datasets in apache spark via python. Using this in tandem with the TensorFlow library offers a foundation for a simple distributed neural network cluster.

2.4.2 Google Cloud Machine Learning

Google is permanently on the forefront of machine learning, and neural networks. Most recently, in 2018, Google released their own Cloud Machine Learning Engine. It claims to build superior models and easily deploy them into production[23] . The exact practices employed in the system are unknown, but they are offering the ability to perform training, predictions and hyper parameter optimisation, on their own distributed neural network clusters to users, at a price per hour.

2.4.3 TensorFlow-On-Spark

TensorFlow-On-Spark is an open source framework developed by Yahoo, and released in 2017, for distributed deep learning on big-data clusters. It brings the scalable architecture of Hadoop and Spark clusters and combines them with the salient features from TensorFlow[24]. This creates a scalable system for distributing training and managing large volumes of data.

2.4.4 Uber Horovod & Michelangelo

Uber attempted an internal alternative to TensorFlow[1] distribution, built on top of the framework. Originally it used a version of a ring-allreduce algorithm published by Baidu[26] , for averaging gradients and communicating these averages to all nodes, in which each of N nodes in the network communicates with two of its peers 2*(N-1) times[25]. The approach also uses Message Passing Interface (MPI) to launch all copies of the TensorFlow program, and then MPI allows the infrastructure for workers to communicate with one another[25]. This simplifies and speeds up the process compared to default distributed TensorFlow. Eventually in 2017 Uber released Horovod, as an open-source framework[25] . In the final release they replaced the ring-allreduce algorithm with NCCL from NVIDIA[27] for collective GPU communication that provides a highly optimized variation of ring-allreduce[28] .

Uber also released its own Machine Learning platform in late 2017 entitled Michelangelo[29]. It was constructed to allow teams to seamlessly build and deploy neural network models at Uber's scale, covering all the workflow, training, evaluation and data management. It uses a mix of open source systems previously discussed such as HDFS, Spark, and TensorFlow.

2.4.5 Current Design Issue

A large issue with all modern distributed neural network platforms is that they rely on hardware owned, maintained and controlled by the network creator. This can be ideal for larger companies as it offers a high degree of security, control, and customisation for the network, but this is not always necessary and limits the potential size of the network depending on the resources the business can delegate. On top of this, large networks are costly with large overheads to keep them running.

By instead relying on how many users are connected and willing to assist, it allows for the potential for far larger networks than normal, with the trade-off of unreliability due to the necessary workers needing to be available. Although this unpredictably could be a possible deal breaker for larger companies, the potential for faster and better training at far reduced overhead cost could be an attractive option for smaller organisations.

Chapter 3.

Design

This chapter details the process by which each aspect of the platform was formulated, beginning from the problem which needed to be addressed, the options that were considered and the approach implemented in the final model.

3.1 Problem Formulation

We begin by creating a breakdown of the problems which must be overcome for the implementation of the platform, and the possible solutions to each.

3.1.1 Dividing the network

The foremost issue when dealing with a distributed system is how to adequately divide the work amongst the participating nodes in the network. Two solutions were explored to deal with this.

The first solution is Model Parallelism. This is dividing the neural network model itself amongst the machines in the network, creating a single model with neurons or whole layers distributed, and each machine responsible for just its own forward pass calculations and back propagation updates and relays the results directly to the relevant nodes or pushes updates to a parameter server.

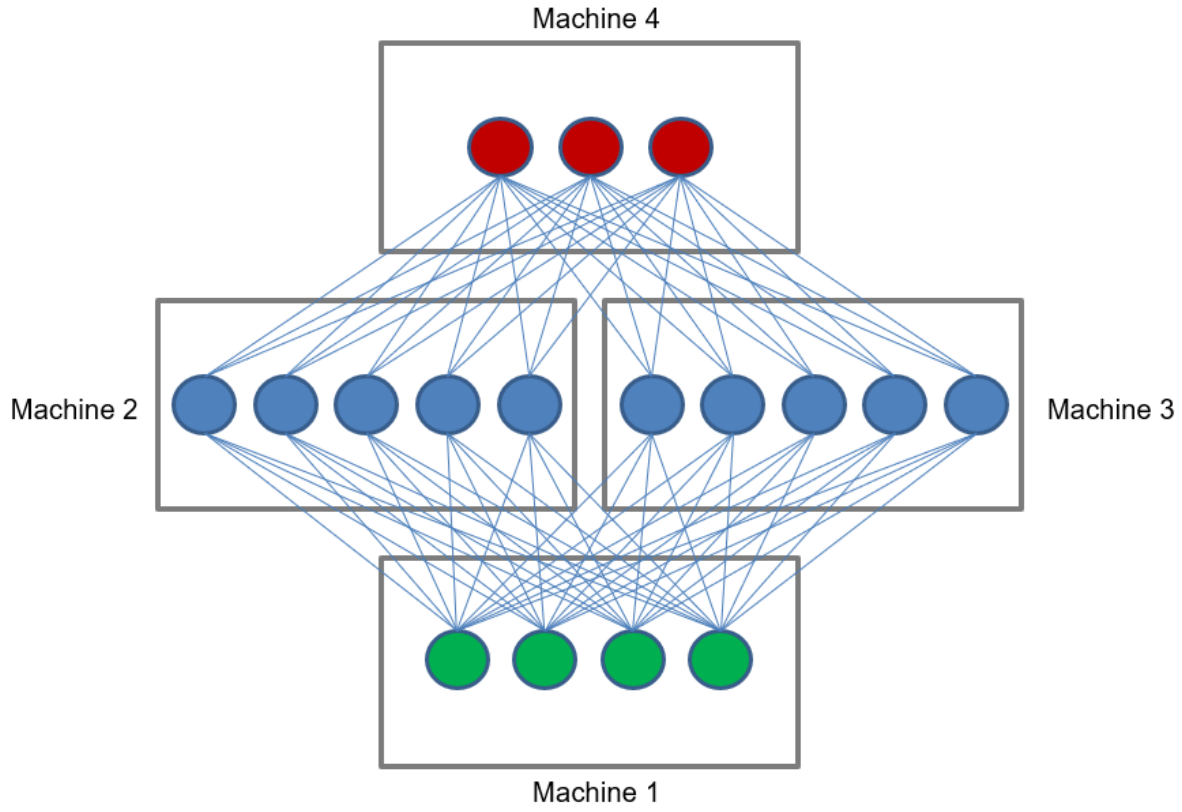


Figure 6: *Simplistic Model Parallel Network*

This approach poses many issues in the loosely coupled network we are designing because we are interfacing with unfamiliar machines, where the hardware and connection speeds of each machine is not reliable. Each stage of this design would require the previous section to have completed its work and passed on the required activation inputs or updated weights, resulting in a large possibility for bottlenecking if just a single machine is slow.

The alternative approach is Data Parallelism. In this method we implement a full neural network model on every machine. The machines each push their updated weights after every batch to the parameter server, where it is averaged with the current weights so far, and the resulting weights returned as the true weights to the worker. This allows for multiple machines to constantly push updates to the parameter server, edging the average weights closer to some local minima of the cost function, and thus cooperatively training the neural network together without relying on any single machine.

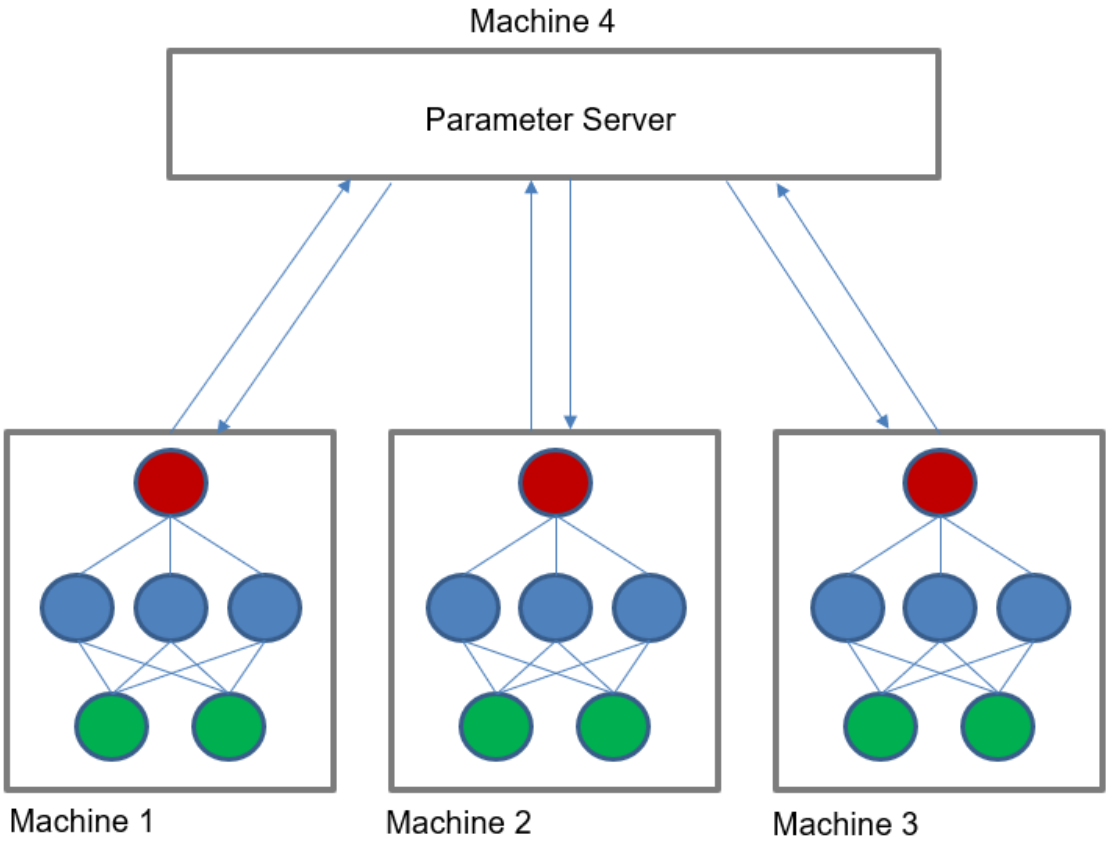


Figure 7: *Simplistic Data Parallel Network*

This approach is not without its issues, as each machine is trusted equally, and a single client pushing grossly incorrect weights could impact the overall effectiveness of the system. Despite this, the lower possibility of bottlenecking, made this a far more attractive option.

3.1.2 Establishing the Network

The first problem which must be overcome is establishing awareness across the multiple remote machines that will form the network. For this there is no way to get around having some central architecture which can act as a waypoint through which these machines learn of each other. The simple solution to this is a database of currently accessible machines along with a matchmaking server to pair machines together and send the relevant addresses of other machines for it to match with, along with the hyperparameters of the network that each node needs to work with to maintain consistency.

Once these addresses and parameters are shared to each machine, it is important that we maintain some method of identification for each node in the network that is consistent across each node for assigning jobs/tasks, data distribution and allocating the correct segmented data to work on.

3.1.3 Distributing & Managing Big Data

Big data is the backbone of neural networks and one of its largest issues. In most circumstances the issue is that it is either time or resource intensive to gather vast amounts of data needed to adequately train a network. In a distributed neural network, a new issue arises as assuming we have this vast amount of data, a method is needed to efficiently share it amongst all the relevant machines in the network.

Single File Server

The simplest solution is having a file server which the host can upload the data to, and workers can fetch the data from. Alternatively, the host itself can act as the file server, and share the file directly to each worker in the network. A single point file server This approach is fine for smaller networks but as the number of workers grow, we get a linear increase in the time to distribute the file across all nodes.

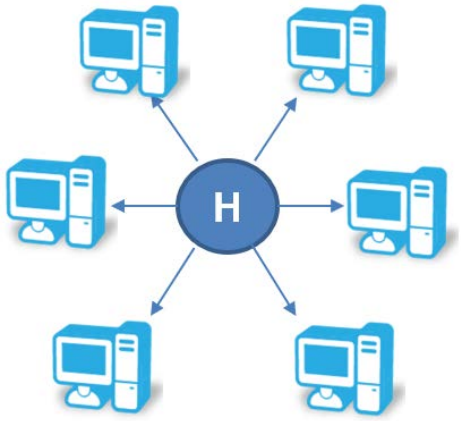


Figure 8*: Single Point File Distribution Architecture*

Hadoop Distributed File System

Built using the Hadoop architecture previously discussed, a simple 3-node Hadoop distributed file system could be implemented, which would provide a more accessible data distribution platform that could also scale much better depending on demand. This would offer a better alternative for being able to both distribute and store large data but requires the platform to host the file cluster, adding another possible point of failure that would limit all new neural networks from being created.

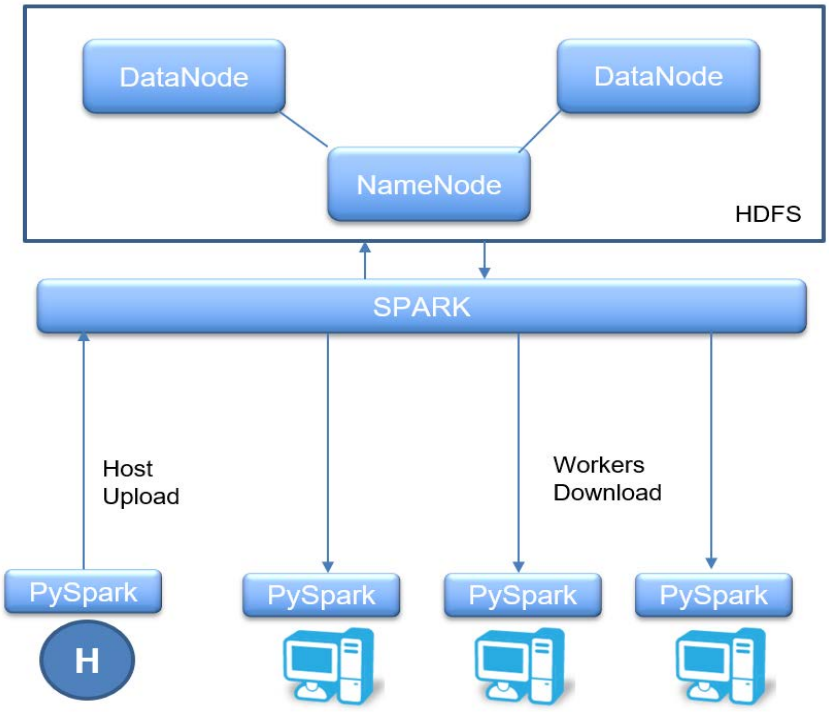


Figure 9*: HDFS Architecture using Spark*

Replication

The best option is to push the data managing and distribution back to the edge and minimise central server interaction as much as possible, although, as previously discussed, having the host distribute the file to each worker was far too slow. The solution was a file replication algorithm, which would allow each worker as it received the file to begin distributing to other workers.

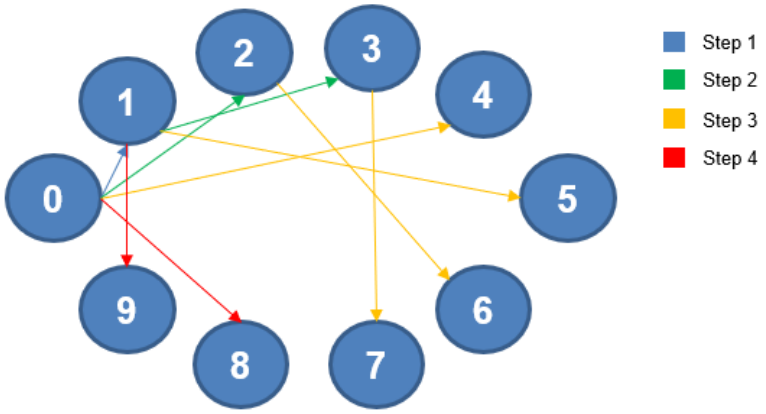


Figure 10: *Visualisation of the replication of data in the network*

Such a method as shown in Figure(10), should allow the system to cope far better with increased number of nodes in the network, and better distribute the data transfer process across available machines. Taking an example network of, 1000 worker nodes, assuming an average transfer speed of 100MB/s across each node, and a dataset 1GB. It would take a single point system over 2 hours to transmit the data individually to each node using a single point file server. In contrast a replication system could theoretically propagate the data in under 2 minutes. The largest flaw in such a peer-to-peer replication method though, is the cascading effect a single node going down would have, resulting in no future nodes in that chain receiving the data.

3.2 Final Design

3.2.1 Design Architecture

Considering the problems and decisions made in section 3.1 Problem Formulation, the final architecture of the design can be broken down into 3 stages, shown in Figure(11).

1. Establishing the network via a central matchmaking server which stores available machines and hosts.
2. Distributing the file via a simple replication algorithm to avoid any more interaction with the central architecture.
3. Begin training the data using a data parallel method with each node independently interacting with a parameter server that manages the current graph of the network.

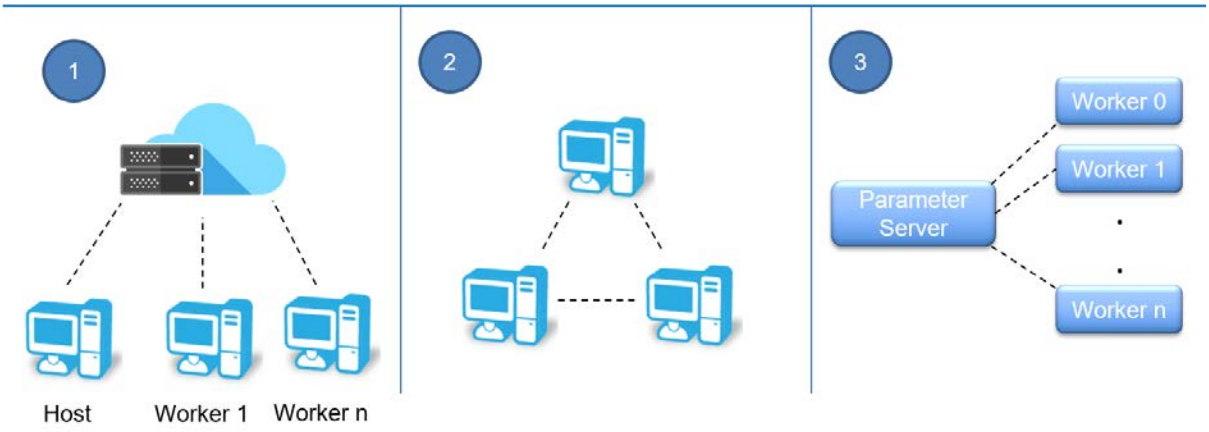


Figure 11: *3 stages of final design*

Chapter 4

Implementation

This chapter details how the previously discussed methods of the design were actually executed in the final model, demonstrating the step by step breakdown of the network formation, data distribution and training processes which happen from the first to last stage.

4.1 Establishing the network

Initial Connection

The process begins with a client establishing a connection to the matchmaking server via TCP socket. As part of this initial connection, the client also supplies a list of information that the server is expecting from the connection; job identifier (Host/ Worker), IP address & port, and various neural network hyperparameters. The client is given a wait command upon joining.

Data and Client Handling

A new thread is created for each client which connects to the server, this pushes the IP address of the current client to either a worker or host list, depending on the job identifier supplied by the client. If the client is a host, the supplied parameters are also stored into a host specific array. Due to the likely high turnover rate of clients, and the fact it is unnecessary to retain any client information once the match making process is complete, a more robust database system is not necessary. This results in every client connection being treated as a brand-new machine, with no memory of any prior connections.

Periodically, using a first-in first-out basis, a host is selected along with a number of necessary workers. These are removed from their respective lists of available clients and moved to a new unique list containing just the IP addresses of each needed machine that will form this new network.

Grouping

Each thread handling an IP address located in this new list is contacted and set to active. These threads then copy the list of addresses from the list, along with the hosts specifications, to a single string. The ordering of the addresses is identical across all client replies, this is important as positioning within this order is used for identifying nodes on the network for file replication and job distribution.

Client Return

Finally, the awaiting client receives a new reply from the server containing each of the IP addresses which will form the network, along with the host specified network hyperparameters. This information is parsed by the client and saved to specific variables. Once complete, the connection to the server is terminated and the client specific thread on the server ended.

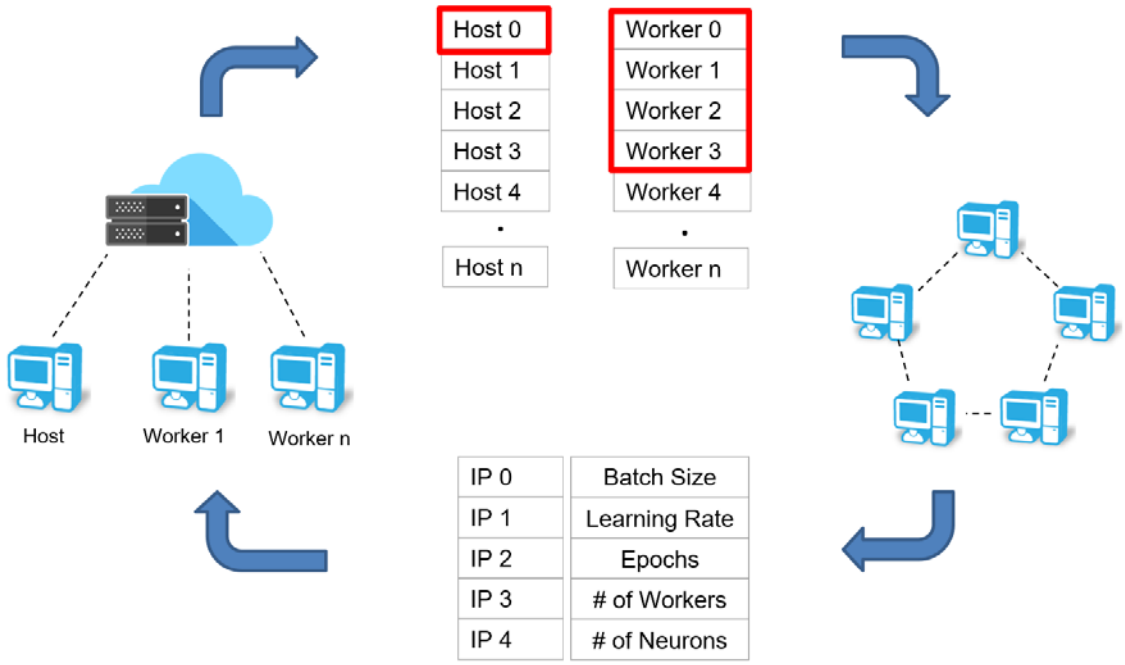


Figure 12: *Breakdown of connection process*

4.2 Data Distribution

Once the client has established all fundamental variables from the server, the next step is to share the original dataset, on the host machine, with each node in the network. The method for doing this, as previously discussed, is a custom replication algorithm, which uses each node once it has received the data as a new host for the data. For this solution it was undesired to perform any polling of workers to see if they have the file, or for there to be any randomness to the process, and thus typical gossip/epidemic algorithms were disregarded. The aim was to achieve the most efficient distribution possible, where the number of nodes transmitting and receiving the file at once was maximised. To do this a route for propagating the data across the network must be established and known by all workers instinctively.

The first step is a simple mapping algorithm, which takes in the number of workers in the network and creates a list of every connection that needs to happen for the data to propagate across the nodes. This is done via a simulation function which initialises a step variable to 1, workers which would have the file at this stage transmit the file to the node which is 2^{step} from the current node position, and this connection is saved to an array. The step value is incremented, and the process repeats until each node in the simulation has the file.

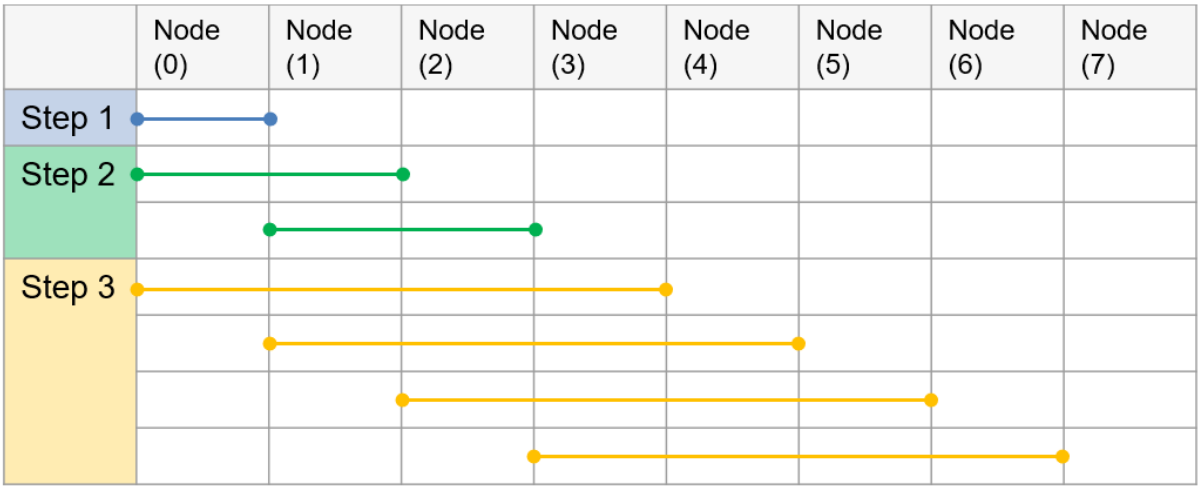


Figure 13: *Simulation of Network Connections*

Then a secondary function takes in the specific node identification number (its position in the original array of addresses) and returns a list of addresses specific to that node

for the data replication process. For example, in the case of Node (1), as shown in Figure(14), there are 3 connections which Node (1) is part of. The corresponding node number for each of these connections is saved to an array, and then the IP of each of these nodes is gotten from the original address list. For every client other than the host, the first IP address will always be the received address from which the node will get the file, and then each subsequent address must be connected with and transmitted the data once the previous connection is satisfied.

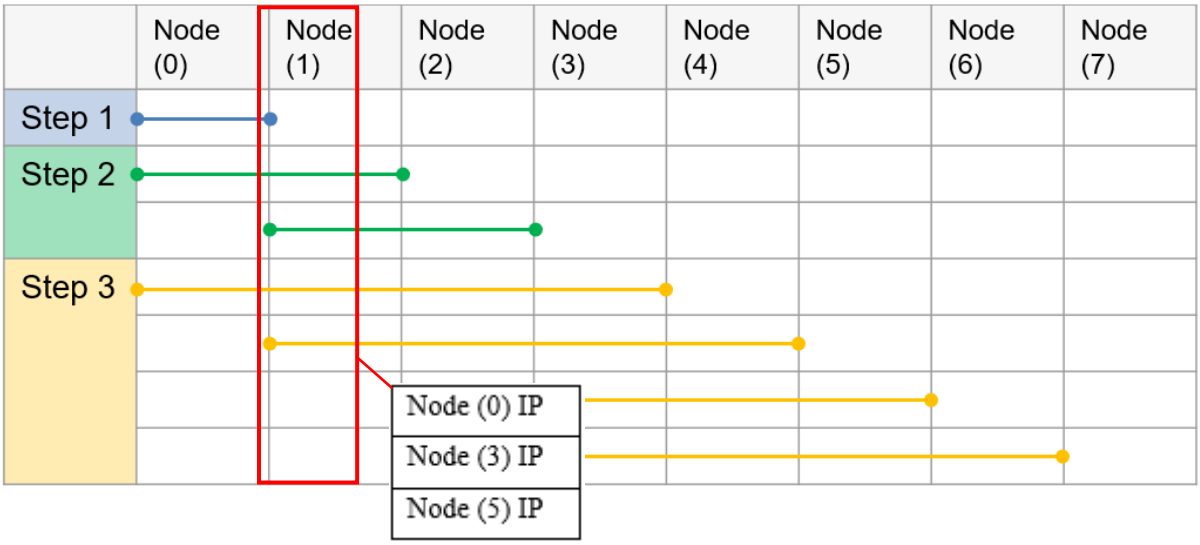


Figure 14: *Isolating connections for Node (1)*

4.3 Training

Finally, once each node has all the necessary parameters, addresses and datasets, our training cluster can be established. The host machine begins by creating the parameter server and a chief worker, worker 0. Each node also creates a respective worker which connects to the cluster with a job ID (J) that is used for distributing the work and worker identification within the cluster. The process will not proceed until every machine, from the original IP list returned by the server, has connected.

Utilising the same hyperparameters as the host, each worker begins training their data on a full local model copy of the neural network. Based on the number of workers in the cluster, the data is split into N + 1 segments. Based on the J value of the node, each node operates on segments J to J+1. This allows each worker to perform its calculations on a smaller fraction of the data, speeding up processing time, while also maintaining an overlap in data to increase model accuracy.

As each node finishes a forward pass through the network for a full mini-batch, the average results are compared with the expected results, and a loss/cost is equated using a cost function. The cost function used is important and all nodes should be using the same one. This could be shared via the matchmaking server along with the other hyperparameters, but in the current platform iteration it is hardcoded using a cross-entropy cost function.

Using this cost, we perform back propagation to update our weights in the model and gradient descent to gradually approach the local minima of the cost function. Our new graph is calculated locally on each worker, and then the differential is shared with our Parameter Server (PS) via gRPC Remote Procedure Call. The PS averages the graph received from the worker with the current graph to gain the new weights / biases of the model, which are replicated back to the worker, and then the process begins again. Every worker in the network is pushing these updates to the PS and receiving back the most up to date variable values for the next pass through, resulting in every worker and the PS consistently getting closer to the local minima.

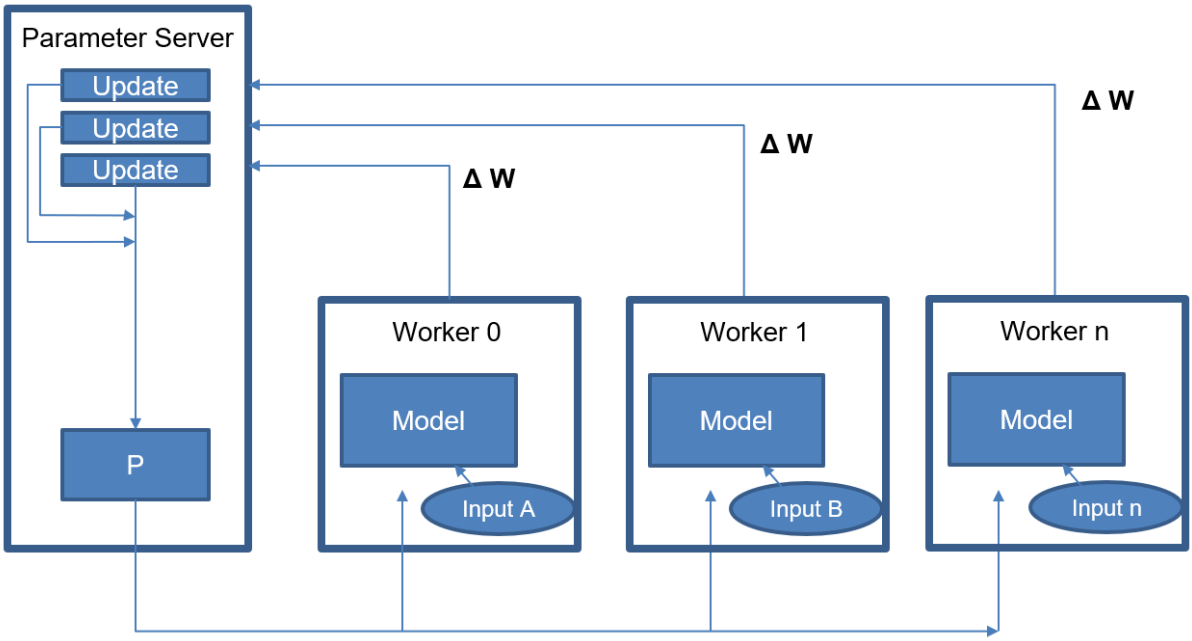


Figure 15: *Breakdown of the Distributed Neural Network*

The system is using asynchronous updates between the workers and the server, where each worker has an independent training loop that will update the graph on the PS and fetch updates as soon as it can without consideration for any other nodes. This is to prevent any bottle necking from slower machines which may be part of the network. As the size of the network increases there is a likelihood that a single parameter server will not be able to keep up with the numerous updates and become a networking or computational bottleneck. In this case, workers can be assigned as extra parameter servers as the network grows, which would then in turn periodically average between themselves. If too many parameter servers are used though, the communication pattern becomes “all-to-all” which may saturate network interconnects. An alternative to alleviate stress on the PS, would be to do synchronous updating where we would wait to have all worker changes before averaging this weight change and only then updating the graph on the PS.

In this particular experiment we are performing our neural network on the MNIST dataset, a commonly used collection of more than 60,000 hand draw numbers from 0-9, on a 28x28 pixel grid. The pixels are each input to the network as a greyscale value, resulting in 784 input and 10 output neurons. The number of neurons in our hidden layer is arbitrary, and in this case 100 neurons are used.

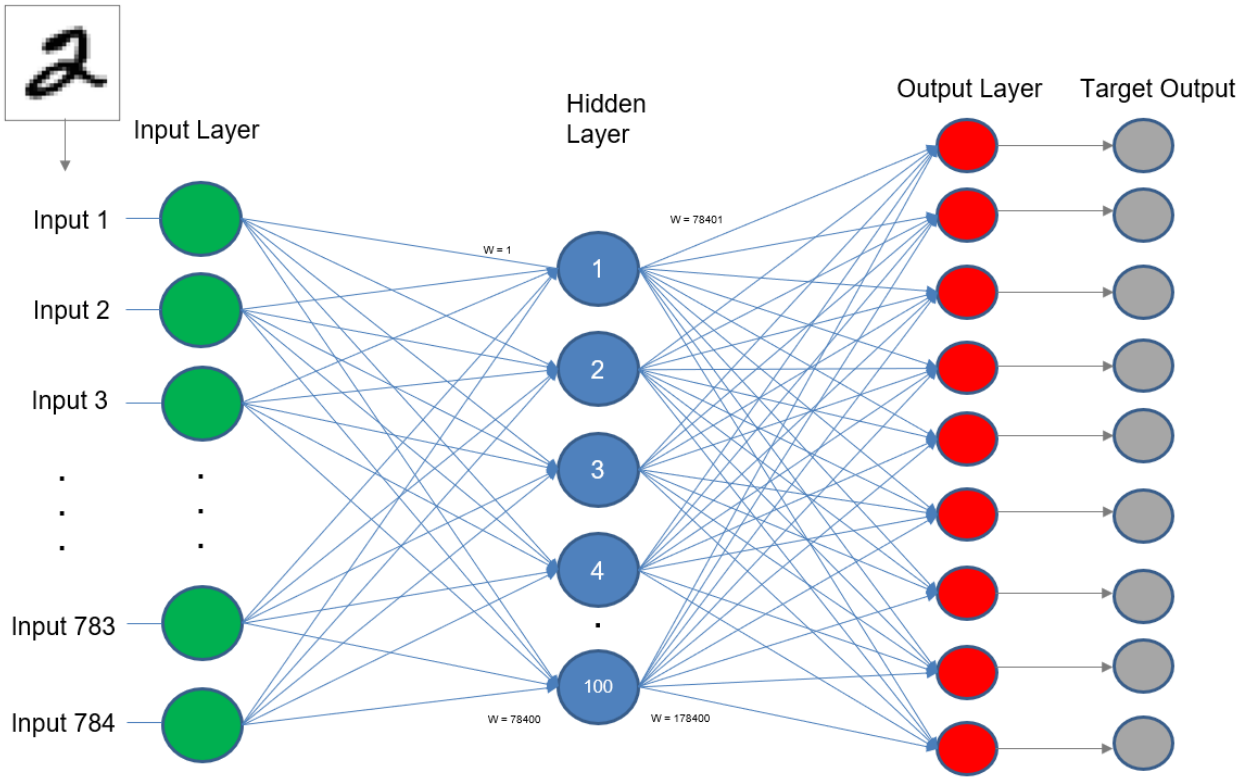


Figure 16: *Single Neural Network Model Example*

In total this results in nearly 180,000 weights which are being updated with each back pass through the network, the result of which are passed to the PS.

4.4 Platform Interfacing

A single intuitive and clear interface for the entire platform, is a key goal of the project. To create a simple Graphical User Interface (GUI) PyQt is used in this project. PyQt is a binding for the QT application framework. The UI itself can be defined through a drag and drop system or through explicit XML. Each component of the interface is given a label which can then be interacted with in the python code.

Sliders are used for varying the neural network hyperparameters, and the user is given simply two options to either 'Host' or 'Work'. Upon clicking the 'Host' button, the current slider values and the file name/path is saved to a config file. A connection is then opened to the match making server as discussed in Section 4.1 (*Establishing the Network*) and the contents of the config file are sent to the server. A similar process occurs when the 'Work'

button is pressed, only the config file is irrelevant, and its contents ignored.

A stop button which terminates all active threads is included, to cease all connections and active training, and finally a large output area is used to inform the user of the current actions being performed and the final result of the training.

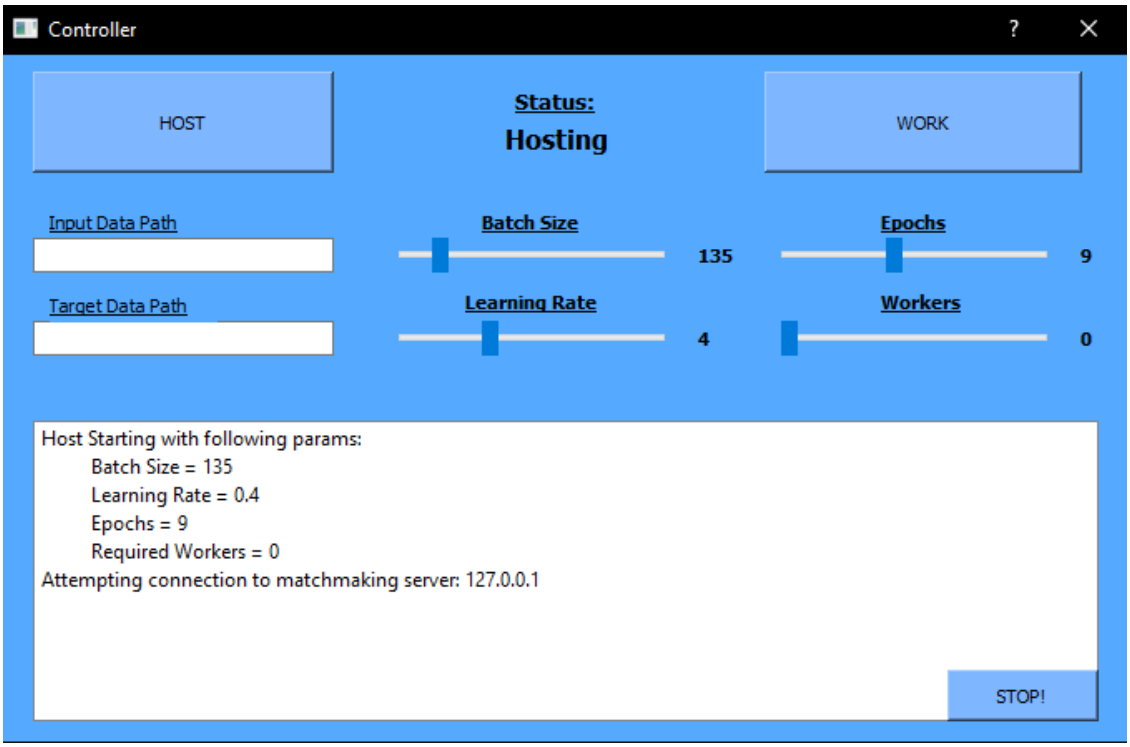


Figure 17: *Platform User Interface*

Chapter 5

Evaluation

This chapter details the main tests that were conducted over the course of this project. Each evaluation begins with a representation of the results observed during the testing stages of the research, followed by a discussion of these results and finally a summary of the overall effectiveness of the model. This section also covers issues which occurred during testing and discussion of metrics which were unable to be evaluated properly.

5.1 Accuracy

The main evaluation metric in determining the overall goodness of a neural network can be assessed by either the overall cost/loss (regression problem) or accuracy (classification problem) that was achieved when the network is verified against the test data. When the distributed platform was tested with varying numbers of workers, each hosting a full dataset instead of a segment, we find a near linear increase in testing performance of the network, shown in Figure(18). This increase in accuracy came at no extra cost to time and was near identical to a single worker duration, when excluding italicisation interval.

Average Accuracy ranging from 0 workers (Only Host) up to 3 additional workers:

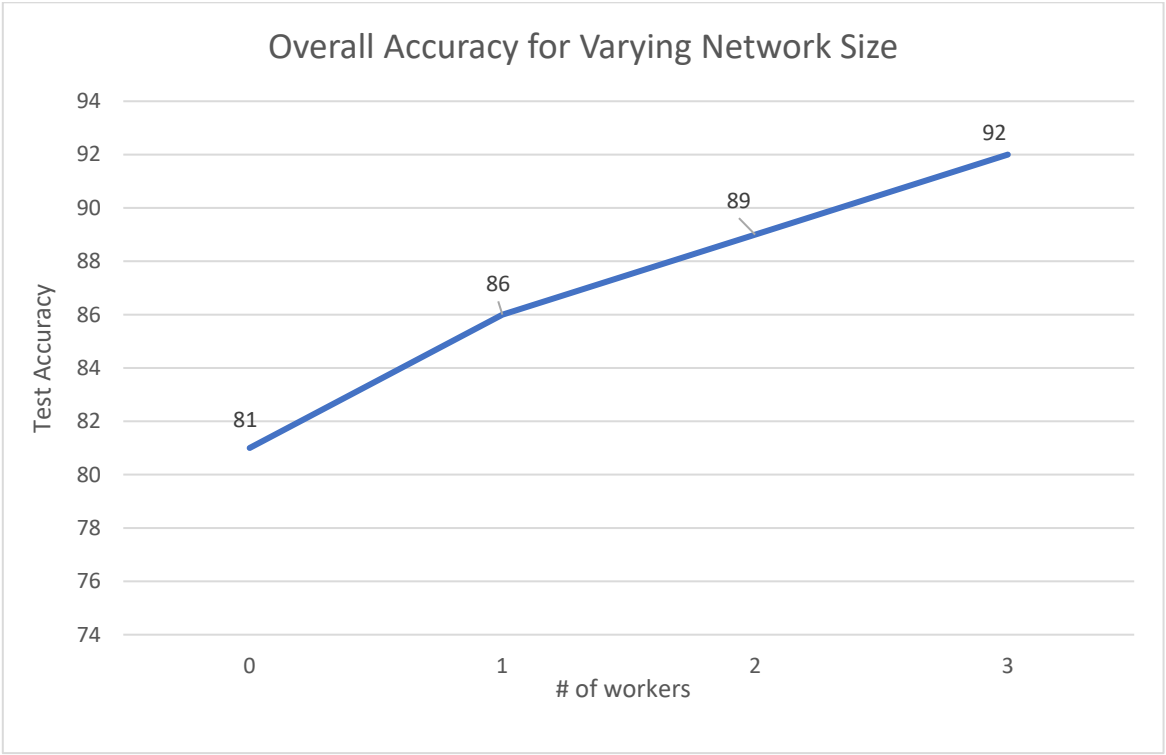
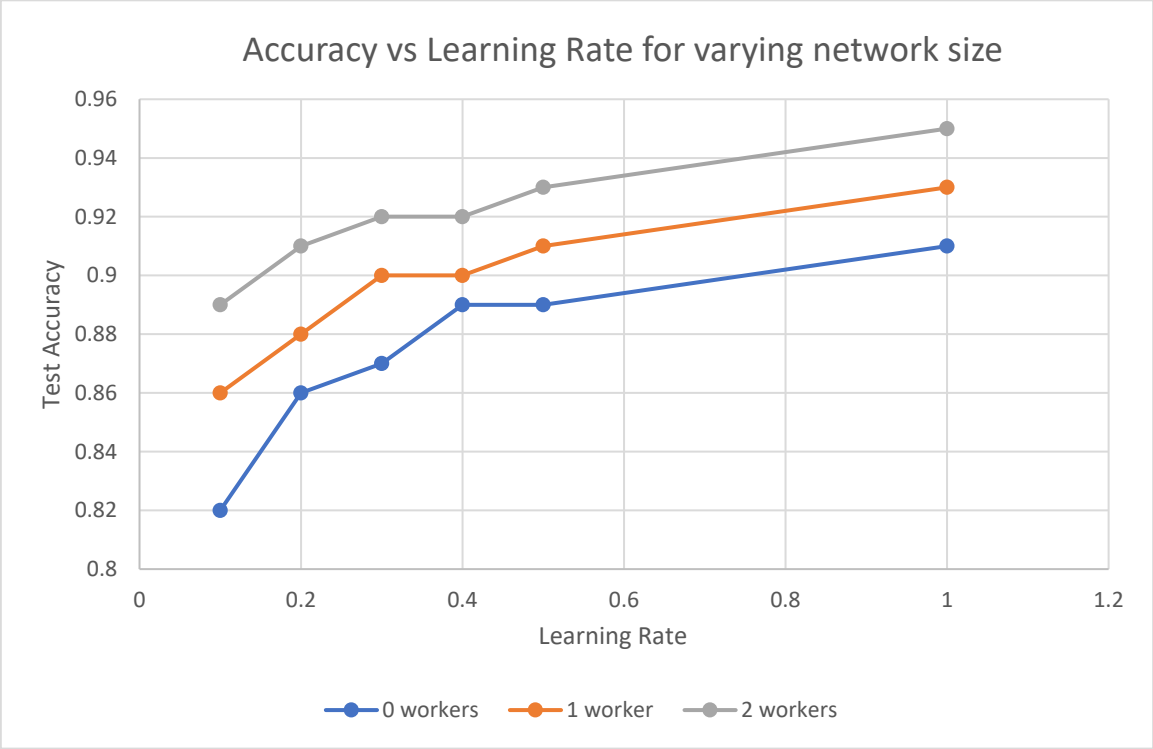


Figure 18: Graph of average network accuracy using default hyperparameters

A single additional worker had the largest increase in accuracy, jumping up on average 5%. Additional workers after this each averaged another 3 % increase in test accuracy. This method is subject to diminishing returns and it would be estimated that eventually overall accuracy increase would be negligible with increased numbers of workers, but the time factor could be decreased at a near linear rate once the network is established.

For this testing a default set of hyperparameters was used in each test case. The next important aspect to question is the effect of the various hyperparameter values and whether the accuracy achieved was more dependent on any particular hyperparameter than another. To test this the network was run through various iterations of changes to each parameter and the results of each case taken and graphed to visualise the impact of each parameter across multiple worker variations.

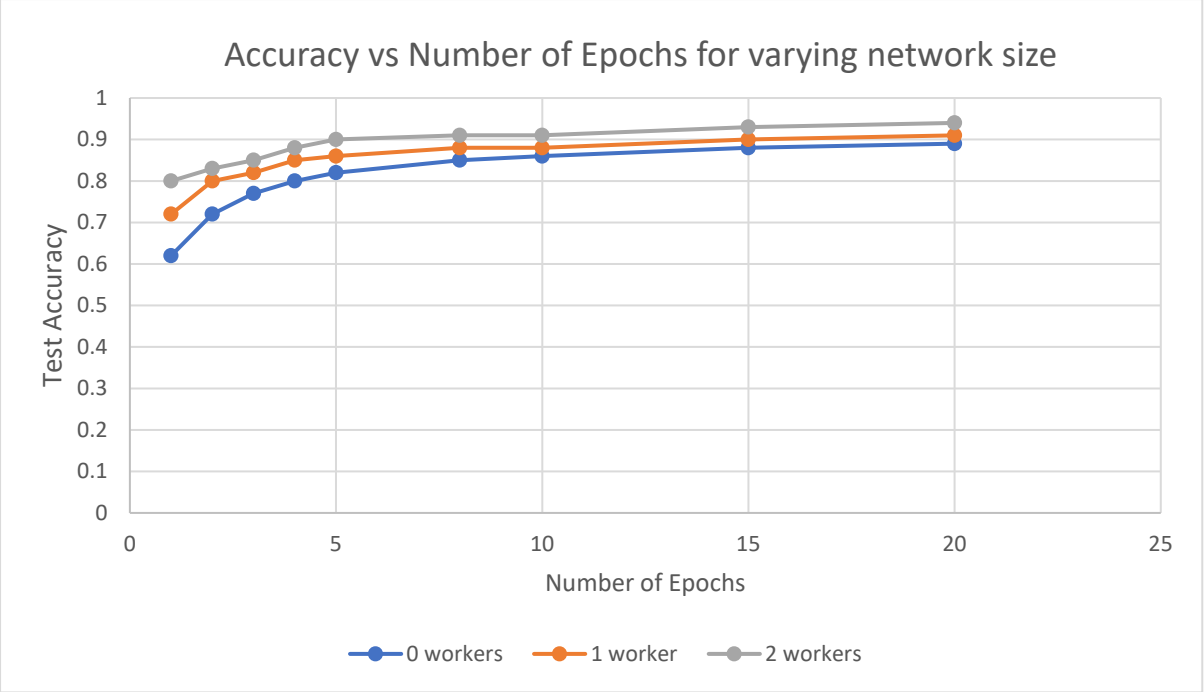
Testing using the varying hyperparameters we achieve the following results:



	Accuracy		
Learning Rate	**0 workers**	**1 worker**	**2 workers**
0.1	0.82	0.86	0.89
0.2	0.86	0.88	0.91
0.3	0.87	0.9	0.92
0.4	0.89	0.9	0.92
0.5	0.89	0.91	0.93
1	0.91	0.93	0.95

Varying the Learning Rate marked a consistent improvement from additional workers on the testing accuracy. The improvement showed little variance depending on the range of learning rates tested, affirming that the increased effect of more workers is neither increased or diminished by the scaling of this hyperparameter.

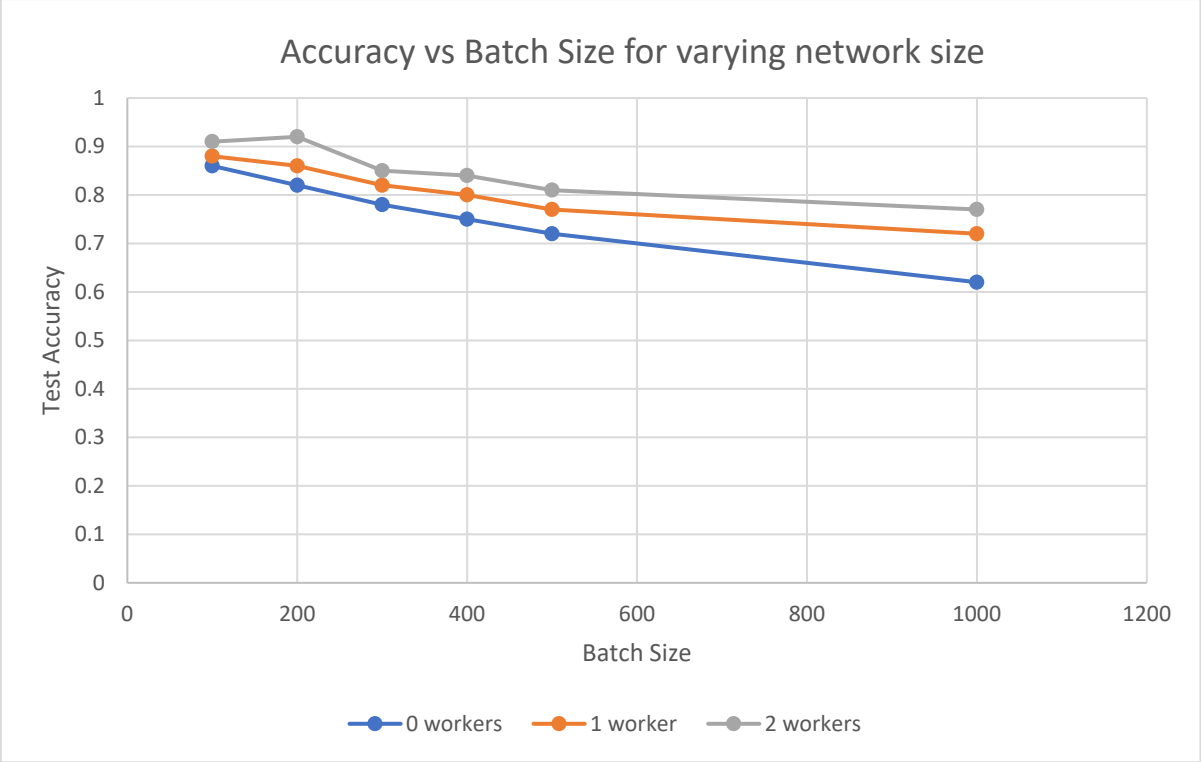
- No Additional Workers Average: 87.33%
- Single Additional Worker Average: 89.66% - Increased accuracy rate of 2.33%
- Two Additional Worker Average: 92.00% - Increased accuracy rate of 2.33%



	Accuracy		
Epochs	**0 workers**	**1 worker**	**2 workers**
1	0.62	0.72	0.8
2	0.72	0.8	0.83
3	0.77	0.82	0.85
4	0.8	0.85	0.88
5	0.82	0.86	0.9
8	0.85	0.88	0.91
10	0.86	0.88	0.91
15	0.88	0.9	0.93
20	0.89	0.91	0.94

Varying the Number of Epochs resulted in an improvement from additional workers on the testing accuracy. As would be expected, there is a larger increase in test accuracy with fewer epochs, which slowly grow to converge as the number of epochs increases. This is due to the case of diminishing returns when it comes to training a neural network which becomes more difficult as it trains, with each pass through the data resulting in more and more minor changes.

- No Additional Workers Average: 80.11%
- Single Additional Worker Average: 84.66% - Increased accuracy rate of 4.55%
- Two Additional Worker Average: 88.33% - Increased accuracy rate of 3.66%



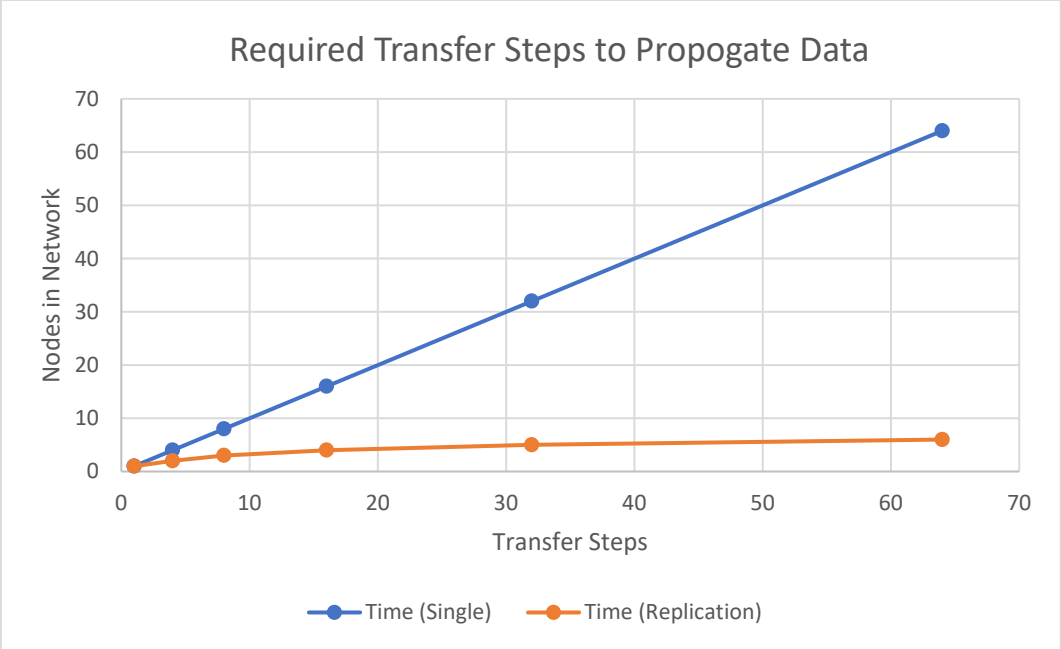
	Accuracy		
Batch	**0 workers**	**1 worker**	**2 workers**
100	0.86	0.88	0.91
200	0.82	0.86	0.92
300	0.78	0.82	0.85
400	0.75	0.8	0.84
500	0.72	0.77	0.81
1000	0.62	0.72	0.77

Varying the Batch Size marked an improvement from additional workers on the testing accuracy. In contrast with number of epochs, a smaller batch size resulted in a lower impact from the increased size of the network, slowly diverging and gaining a larger margin as the batch size increases. A larger batch size means the network is pushing updates to the gradient more infrequently, and as such having more workers contributing to the gradient steps would be expected to increase the accuracy as seen.

- No Additional Workers Average: 75.81%
- Single Additional Worker Average: 80.83% - Increased accuracy rate of 5.02%
- Two Additional Worker Average: 85.00% - Increased accuracy rate of 4.17%

5.3 Data Distribution

The replication method implemented as part of the platform is simple in execution and performs far better than any kind of single point file distribution. This method only offers a noticeable increase with network sizes with 4 or more nodes. Due to an inability to functionally test this algorithm across a larger network, the following data is purely theoretical.



5.4 Evaluation Limitations

Due to the experiments being conducted on a single machine, any time metrics obtained are likely inaccurate representations of what would be present in a real-world scenario and thus were cut from this paper. The largest bottleneck in this system is the rate at which data can be transferred between workers and the parameter server. In large internal networks, such as the ones in Google, all machines are hardwired using InfiniBand cables for overcoming this issue. Due to this being an external network over IP much larger latency should be expected.

It should also be noted that due to random primary weight initialisation, which of the possible local minima of the cost function is converged on by the network can vary. This results in inconsistent final accuracy scores between different runs of the network.

Chapter 6.

Conclusion

6.1 Design Applications

In its current state the platform could be used to make use of unused or idle machines and allow them to volunteer their local processing power to help a host train a neural network, potentially opening up the complexities of neural network training to anyone despite their situation and without the need for expensive hardware. This could also allow businesses or universities with underused machines during off peak hours, to use those machines to help train neural network projects.

6.2 Future Work

Due to time constraints during the period of this research, not all aspects of the platform goals were able to be realised. As such there are additional components and areas in which the platform could be improved with future contribution.

6.2.1 Fault Tolerance

Currently in the model there is no contingency for if a node goes down during the file replication process and would result in the whole process needing to begin again. This is because of the method by which the replication happens, there is no polling or testing to see if a node still exists, the network just assumes that each node is present and performing as it should. A potential method to solve this is to allow the network to propagate as normal, and after which any nodes which do not connect to the network because they did not receive the file, could then be polled to see if they are active and then directly transferred the data via the host.

6.2.2 Security

The parameter server currently values the information from each worker identically, and the weight changes provided by each machine are assumed to be valid updates. This causes issues if you have a bad actor in the network committing fraudulent or exaggerated values to the parameter server. A possible solution is to flag any updates which deviate too far from the current average or allow for a legitimacy score held by the parameter server for each worker, which could increment as valid changes are committed and act as a weighting to the information supplied by the worker.

6.2.3 Host Targeting

It would be necessary to implement an option for client workers to specify a target host IP from the user interface. This would allow people to aid certain, businesses or causes which they are interested in. For example, someone who wants to help a game company they enjoy could train a neural network for better match making or flagging toxic commentary. Or an individual could lend their processing power to specific medical research which is analysing trends of symptoms and treatments in cancer patients.

Allowing users to do this is the greatest way to get attention for the platform as it would be in the hosts interest to get users on board, and thus allow the platform to gain traction with users. Having a large collection of users to draw upon is paramount to the platform performance, and as such implementing this would be integral.

6.2.4 Dynamic In-Progress Network Connection

A key shortcoming with the current design is the initialisation time when setting up the network and the requirement that all workers are present, and once begun no new workers are able to be added to the network. A necessary addition to the platform in the future would be to allow new workers to join networks in progress, fetch the most up to date weight values and begin contributing to the training.

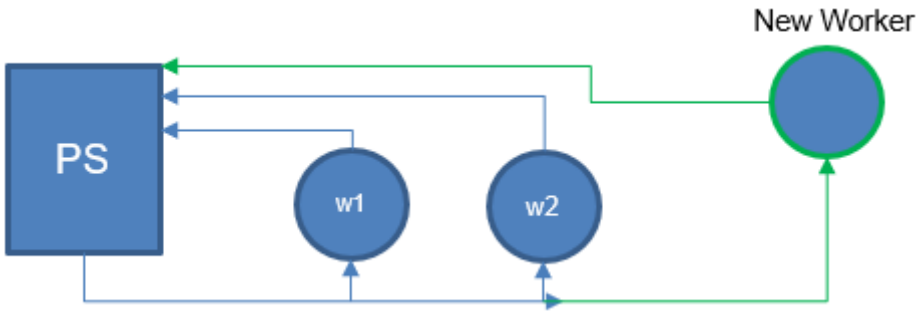


Figure 19*: Visualisation of new worker added to network in progress*

This would work especially well in tandem with the targeted hosting, as it would allow a host to set up a training network, post the relevant IP and port for users to see, and gain a multitude of workers over time, which are free to drop in and out of the network.

6.2.5 Further Testing

Finally, the overall scope of testing for the project was limited compared with what is necessary to fully evaluate such a platform. Perhaps devoting more time to testing would broaden the number of areas of evaluation and gather more results on the performance of the platform under a variety of different parameters and environments. In particular, scaling the number of workers and seeing how the model holds up, or implementing other neural networks models such as Deep, Recurrent or Convolutional Neural Networks.

6.2.6 Mobile Development

As smartphones and other mobile devices are constantly improving in their capabilities, the ability to harness this local processing power, which is idle the majority of the time, has huge potential in the broader scope of distributed processing. Adapting the platform to be used for mobile devices, possibly through the use of the Keras library[30] in python would be great addition, but the approach to data management and distribution would need to be reconsidered. A possible solution to this, which could be adapted to the platform as a whole, is that currently ever worker is transmitting a full copy of the entire dataset. Instead, it could be implemented to only transfer the relevant data segments that the current node, and its future branches, require. Using this method, the total amount of data being sent is reduced,

and each transfer step is smaller as the data propagates. This would reduce overall download requirements, speed up transfer time, and alleviate some storage concerns.

6.3 Closing Comments

The final design and implementation was able to successfully create a single application which could be used to connect a dynamic number of computers and allow them to create a scalable network for training a neural network cooperatively across heterogeneous machines. There were some shortcomings in the final design implementation due to time constraints of the research duration and time spent getting a functional model, but the final design has been able to lay the ground work and serve as proof of concept for the design model. Overall this design is a good stepping stone toward the final goal, but there are still many additions which would need to be added to fully realise the potential for the platform.

Bibliography

[0] Bernard Marr, *The Amazing Ways Googles Uses Deep Learning AI*, available at: https://www.forbes.com/sites/bernardmarr/2017/08/08/the-amazing-ways-how-google-uses-deep-learning-ai/#6475d8f03204

[1] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015. available at: tensorflow.org.

[2] R.L. Adams, *10 Powerful Examples of Artificial Intelligence in Use Today*, available at: forbes.com/sites/robertadams/2017/01/10/10-powerful-examples-of-artificial-intelligence-in-use-today/#57167884420d

[3] UC Berkeley, *Search for Extra-terrestrial Intelligence (SETI),* available at: setiathome.berkeley.edu

[4] Pande Lab Stanford*, Folding at Home*, available at: http://folding.stanford.edu

[5] Google, *gRPC Remote Procedure Call*, available at: grpc.io

[6] Nikolaev, *Single-Layer Perceptrons*, available at: homepages.gold.ac.uk/nikolaev

[7] Phylliida, *Comprehensive list of activation functions in neural networks with pros/cons*, available at: stats.stackexchange.com/q/154877

[8] Suryansh S, Gradient Decent: All You Need to Know, available at: hackernoon.com/gradient-descent-aynk-7cbe95a778da

[9] Robert Hecht-Nielsen, *Theory of the Backpropagation Neural Network*, available at: sciencedirect.com/science/article/pii/B9780127412528500108

[10] David E. Rummelhart, Geoffrey E. Hinton, Ronald J. Williams*, Learning representations by back-propagating errors*, available at: nature.com/articles/323533a0

[11] Chris Thornton, Frank Hutter, Holger H.Hoos, Kevin Leyton-Brown, *Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms,* available at: www.cs.ubc.ca/labs/beta/Projects/autoweka/papers/autoweka.pdf

[12] Olga Davydova, *7 Types of Artificial Neural Networks for Natural Language Processing*, available at: medium.com/@datamonsters/artificial-neural-networks-for-natural-language-processing-part-1-64ca9ebfa3b2

[13] Ian Goodfellow, Yoshua Bengio, Aaron Courville, *Deep Learning*, available at: http://www.deeplearningbook.org/

[14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio, *General Adversarial Networks*, available at: https://arxiv.org/abs/1406.2661

[15] Apache Hadoop, software available at: http://hadoop.apache.org/

[16] Apache Spark, software available at: https://spark.apache.org/

[17] PySpark, software available at: github.com/apache/spark/tree/master/python/pyspark

[18] Jose Marques, Gabriel Falcao, Luis A.Alexandre, *Distributed learning of CNNs on heterogeneous CPU/GPU architectures*, available at: https://arxiv.org/abs/1712.02546

[19] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Andrew Y. Ng*, Large Scale Distributed Deep Networks*, available at: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/40565.pdf

[20] Rohan Anil*,* Gabriel Pereyra*,* Alexandre Passos*,* Robert Ormandi*,* George E. Dahl*,* Geoffrey E. Hinton, *Large scale distributed neural network training through online distillation*, available at: https://openreview.net/forum?id=rkr1UDeC-

[21] Geoffrey Hinton, Oriol Vinyals, Jeff Dean, *Distilling the knowledge in a neural network*, available at: https://arxiv.org/abs/1503.02531

[22] Priya Goyal Piotr Dollar Ross Girshick Pieter Noordhuis ´ Lukasz Wesolowski Aapo Kyrola Andrew Tulloch Yangqing Jia Kaiming He, *Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour*, available at: https://research.fb.com/wp-content/uploads/2017/06/imagenet1kin1h5.pdf

[23] Google, *Cloud Machine Learning Engine*, available at: cloud.google.com/ml-engine/

[24] Yahoo, *TensorFlow-On-Spark*, available at: github.com/yahoo/TensorFlowOnSpark

[25] Alex Sergeev, Mike Del Balso, *Meet Horovod: Uber's Open Source Distributed Deep Learning Framework for TensorFlow*, available at: https://eng.uber.com/horovod/, software available at: https://github.com/uber/horovod

[26] Baidu, *Bringing HPC Techniques to Deep Learning*, available at: http://research.baidu.com/bringing-hpc-techniques-deep-learning/

[27] NVIDIA, *NVIDIA Collective Communications Library*, available at: developer.nvidia.com/nccl

[28] Alexander Sergeev, Mike Del Balso, *Horovod: Fast and Easy Distributed Deep Learning in TensorFlow*, available at: https://arxiv.org/abs/1802.05799

[29] Jeremy Hermann, Mike Del Balso, *Meet Michelangelo: Uber's Machine Learning Platform*, available at: http://eng.uber.com/michelangelo/

[30] Keras, *Keras: Python Deep Learning Library*, software available at: https://keras.io/

Note: All links have been accessed and found available on the 15/05/2018