RingTMP: Locally Distributed Machine Learning on Low Power Devices

Prifysgol Abertawe
Swansea University

Christopher Hopkins

Department of Computer Science
Swansea University

This dissertation is submitted for the degree of
Bachelor

April 27, 2021

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 40,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 100 figures.

Christopher Hopkins
October 2020

Abstract

More data than ever is being produced by low power devices such as smart phones and Internet of Things (IoT) devices at the network edge. The data being produced is so enormous it would be infeasible to send it to a centralised location. Instead models can be trained from data distributed across multiple edge nodes, with machine learning algorithms being performed locally. In this paper I explore training multiple low power devices using a new distributed machine learning paradigm *RingTMP*. This paradigm focuses on low power usage and power efficiency while having the capacity for larger models than comparative systems.

Contents

1 Introduction **1**

1.1 Motivation and Context 1

1.2 Overview 2

2 Literature Review **4**

2.1 Brief Introduction to Machine Learning and Neural Networks 4

2.2 Limited History of Distributed Machine Learning 5

2.3 Model and Data Parallelism 5

2.4 Model Consistency 6

2.5 The Communication Issue 7

2.6 Low Power Hardware, IoT and Mobile Computation 8

3 Problem Description **10**

4 Implementation **11**

4.1 Tooling 11

4.2 Neural Network Implementation 12

4.2.1 Initialisation 12

4.2.2 Forward Propagation 14

4.2.3 Cost Function 16

4.2.4 Back Propagation 16

4.3 Parameter Server 21

4.4 RingTMP 23

5 Results **28**

5.1 Accuracy 29

5.2 Timings 30

5.3 Scalability 31

5.4 Communication 31

5.5 Summary 32

References **33**

Appendices **37**

A Appendix 1 **38**

A.0.1 Code Listings 38

B two **40**

1 Introduction

1.1 Motivation and Context

Machine Learning has become an invisible but ubiquitous part of modern life, and is being used in a plethora of fields and industries. The uses of this technology range from dystopian facial recognition [1] to lifesaving diagnoses [2] and many more purposes besides. Machine Learning leverages existing data to train machine learning models in order to perform a task or find patterns that previously only a human could. The key difference between machine learning and conventional analysis software is that the data itself is used to develop the model. Therefore the quality and quantity of the data can affect the effectiveness of a machine learning model.

As the amount and complexity of the data we are collecting increases, so does the size and complexity of the machine learning models we use to make sense of our data. For example the Internet Archive as of 2020 contains over 70 petabytes of data, while labeled datasets such as AViD have video data in the order of terabytes. [3] We are now reaching a point where the limiting factor of creating a machine learning model is not the data, but the machine learning algorithm itself.

This problem is this. Machine learning models are getting very, very large. For instance GPT-3 the largest NLP model ever trained contains 175 billion parameters. [4] And efforts are being made to create models with trillions of parameters. [5] We have reached the point where its no longer viable to train some machine learning models on a single machine. [6] This is because we have more data and larger models, but the algorithms used to train these are inherently sequential and difficult to parallelise.

The popular current solution is to use a parameter server model. In brief the paradigm is made of two different types of components. The parameter server and the workers. The parameter server holds the global parameters of the model. Workers each hold a separate set of training data and are given the model parameters by the parameter server. The workers then use an iterative ML algorithm to modify the parameters. The modified parameters are then sent to the parameter server where they are aggregated, the global model parameters are updated and the cycle continues until the model has converged on an answer.

While this method has many benefits and is certainly faster than training using a single machine, it has two key limiting factors. First, every worker must communicate with a single parameter server, this limits scalability as eventually the network bandwidth becomes saturated severely impacting performance. [6] Secondly many parameter server models require the whole model to be replicated within each node. [7] This means that all of the workers must be machines of similar specification, else the higher performance machines will spend too much time idle.

In this paper I will outline an alternative distributed machine learning framework: *RingTMP*. RingTMP is a Ring Topological Model Parallel distributed machine learning framework focusing on optimising distributed gradient descent. This is a novel design drawing in inspiration much research but particularly from the STRADS and DistBelief machine learning frameworks. [8, 9]

I believe my distributed framework may have some advantages over the current paradigm, these briefly are:

- There will be less communication between nodes
- A Potential for larger communication bandwidth between nodes
- My framework will be able to hold larger models, given the same resources

My aims more specifically for this project are to:

- Create a prototype RingTMP framework.
- Create a parameter server model framework.
- Demonstrate less communication between nodes
- Demonstrate that RingTMP is at least as scalable than a generic parameter server
- Demonstrate RingTMP can train neural networks to at least the same accuracy in at least the same amount of time as an equivalent parameter server

1.2 Overview

This document is split up into the following sections:

- **Introduction** Current section. Introduces the project and its aims.

- **Literature Review** Presents related research material in similar applications and areas.
- **Problem Description** Description of the problem aiming to be solved.
- **Implementation** Details of the implementation of the solution.
- **Results** Experimental results and analysis
- **Reflection** A small section reflecting on the project
- **Conclusion** Summary of the project and the paper.

2 Literature Review

2.1 Brief Introduction to Machine Learning and Neural Networks

To first understand Distributed Machine Learning you must first understand the fundamentals of machine learning and neural networks.

There are many machine learning methods some requiring training data which we call supervised and some being able to find patterns in data without being given solutions called unsupervised. [10] An example of a supervised system may be predicting house prices, using multiple factors about each house (market data, geographic area, square footage etc.) to come to a conclusion about what the house could sell for. This would be trained using data of previously sold houses to predict current ones. An example of an unsupervised system could be identification of new plant species. This could be done by taking as many features of a plant as possible, then apply a clustering algorithm to see if there are two distinct clusters in the data. If there were then that would suggest two different plant species. Neural networks tend to focus on supervised learning and use a form gradient descent called Stochastic Gradient Descent.

Many machine learning algorithms use a cost function to measure how well or badly they are solving a problem, these algorithms use parameters which are internal variables of a machine learning model and define how they solve the problem. If you map $costFunction(x)$, where x is the model parameter, for every x value. Then a graph will be produced $y = costFunction(x)$, the lowest point on the graph will be the global minimum. There may be other troughs higher than the global minimum these are called local minimums. A global minimum represents the lowest value of the cost function which indicates the parameter values produce the best solution for your problem. Initial model parameters are often randomised, which likely means they will start at a high point on the cost function graph, the goal is to get to the lowest point possible. To do this you must *descend* down the *gradient* to a local minima, the algorithm that does this is called gradient descent for that very reason. This often happens in little steps after the observation of each piece of data. However it is computationally expensive to step down the gradient after each example. It is more efficient to calculate the average step of a randomised selection of data. This is know as Stochastic Gradient Descent.

Neural networks are structures that can perform multi-variable gradient descent when provided with training data. Neural networks are comprised of layers of interconnected neurons in a lattice like structure. Each neuron holds parameter information the adjusting of which through gradient descent leads to the solving of a problem through reaching the local minimum of the cost function. These structures can then be placed in a parameter server and run in a distributed fashion, if desired.

2.2 Limited History of Distributed Machine Learning

One of the first pieces of research into distributed machine learning was 'Distributed Inference for Latent Dirichlet Allocation' in 2008 [11] One of the first instances of distributed machine learning was used to categorise New York Times articles using Latent Dirichlet Allocation (LDA), which identifies the affiliations words have to certain topics. While the paper focused on parallelising the algorithm and running them over multiple artificially isolated cores the results showed that distributed machine learning could have scalability and didn't impact the rate of convergence of the model significantly. This was followed by a paper by Jia et al. [12] which produced much faster results than its predecessors by using memcache layer in every machine, every machine would message every other machine with updates of its local parameters to create an approximate global state, it was mentioned in passing that arranging the nodes in a star topology and caching the values that passed through it could make the system more scalable. After this followed a cambrian explosion of work in this area [9, 13–15] culminating in 2014 when the parameter server as it is known today [6] was produced. This parameter server is highly sophisticated and flexible, accommodating the difference in hardware components while spending more on computation and less time waiting.

2.3 Model and Data Parallelism

When creating distributed machine learning models there two different methods for distributing training, model parallelism and data parallelism. These two methods are not mutually exclusive and can be used in conjunction with one another, such as in DistBelief. [9]. Model parallelism is when model parameters are split between the nodes. As data parallelism is when the data is split between the nodes. [16] Often with model parallelism the whole set of training data is passed through each node. While in data parallelism its common for each node to hold the whole machine learning model.

The key advantage of model parallelism is that machine learning models can be far

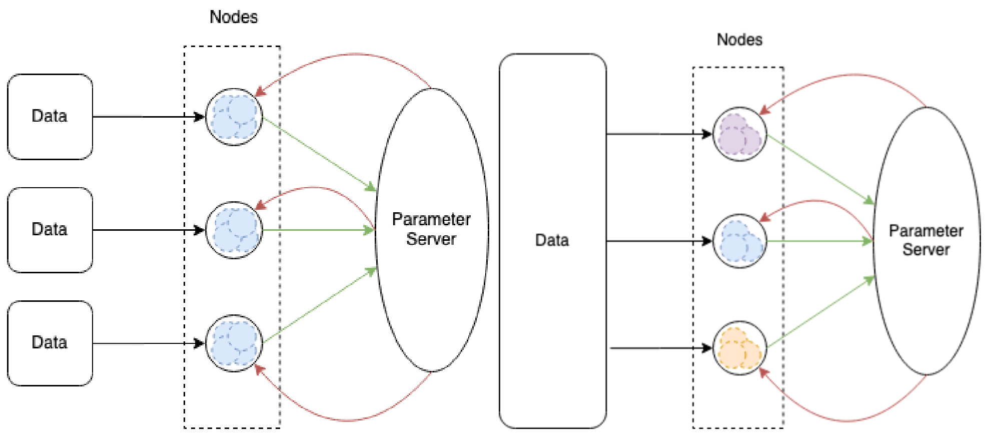


Figure 1: Left: Data Parallelism. Right: Model Parallelism. In both diagrams the green lines indicate local parameters being sent to the parameter server and red lines indicate parameters being sent to the worker nodes.

larger as they no longer have to sit on one machine. However this one great advantage comes with some disadvantages. Some parameters may take more time to converge than others, this means that at times some nodes may be idle while others are still converging, so the spread of computation is not equal or efficient. [9] Because some parameters converge at different rates a scheduler can be used, which does improve model convergence. However this requires more computational overhead and communication and reduces iteration throughput. [8]

Data parallelism has the benefit that data throughput can be very large, making processing using this method very fast. However with more nodes the communication overhead increases as the nodes must communicate the changes in their model parameters to each other. [17]

2.4 Model Consistency

Both of these parallel paradigms still function on the basis of a strictly iterative model, where communication is the limiting factor. Creating parameter server similar to how Googles map reduce algorithm is implemented [15] is still in some sense sequential. The next iteration can't continue until all workers have responded with their updated parameters, and each worker must send its results back no matter how significant its work is. Relaxing these restraints in specific ways have been shown to produce faster training times, while still converging on a model just as accurate. Especially when time, not data is at a premium. [14]

Partially relaxing the iterative restraint of the parameter server and instead using a bounded delay, dramatically decreases the wait times of workers to almost 0. A bounded delay allows workers to operate on slightly stale parameters. Unintuitive this allows for faster convergence than the sequential model, as it can iterate over the data faster and learn almost as much from each iteration. [14].

The other restraint that can be relaxed is the requirement to always send updated weights back to the parameter server. Many training examples don't dramatically change the parameters. Therefore only the training examples that cause the parameters to change significantly need to be sent back to the parameter server. This also allows for greater scalability, as each worker only communicates significant updates to the parameter server, meaning more nodes can be used before saturating the network. [14].

2.5 The Communication Issue

Distributed neural networks must communicate with each other in some way in order to work together. This needs to be formalised to be able to measure the efficiency of our machine learning system. Parts of this reasoning already appears in these papers too. [18,19]

If we consider how a neural network operates if we were to run it on a single node, we could characterise its computation as such:

$$TIME = I_A(\epsilon) \times T_A \tag{1}$$

Where $I_A(\epsilon)$ is the number of iterations of the algorithm A it takes to reach accuracy ϵ and T_A is the time of each iteration of the algorithm. Here maximising the convergence per iteration or decreasing the time an iteration takes will decrease the runtime of the algorithm. In a distributed setting this equation changes to this:

$$TIME = I_A(\epsilon) \times (c + T_A) \tag{2}$$

In this equation we have the added variable c, this represents time taken for communication per iteration. In a distributed setting this will always remain above a non trivial amount of data. Unfortunately the majority of machine learning algorithms use a stochastic method which means a very large number of iterations ($I_A(\epsilon)$) that take a short amount of time (small T_A). You can see that no matter how small c is there will be a significant impact of the time taken. In fact with a naive approach of communication each iteration almost certainly $c > T_A$.

However this view doesn't take into account the possibility that communication could happen at the same time as an iteration. For example imagine a pipeline of nodes where each nodes performs an iteration but can communicate its previous iteration to the next node at the same time. Then the time taken could be described like so:

$$TIME = I_A(\epsilon) \times max(c, T_A) \tag{3}$$

Here you can see that if you can find a way of making the communication time equal to the time per iteration. Then c would have a negligible effect on the equation.

2.6 Low Power Hardware, IoT and Mobile Computation

Historically machine learning algorithms have been focused on high model accuracy through large models and vast amounts of training data, energy consumption and efficiency has rarely been taken into consideration. However with the rise of Internet of Things (IoT) devices and the established ubiquity of smart phones more data than ever is being produced. Soon this data generation with exceed the capacity of the internet, and experts estimate that over 90% of data will be stored and processed locally. [20] By extension this means machine learning algorithms will have to be performed locally too. This introduces some challenging issues. Modern machine learning algorithms require vast computational power and large amounts of data. Local devices don't have the capacity to hold large data sets or the power to compute large machine learning models in a viable amount of time, while many of them are also battery powered so power consumption becomes another issue.

A solution to this is to massively distribute the model over multiple decentralised nodes. The level of distribution is even greater than that of centralised compute clusters. In this solution each device computes a model using its own local data, infrequently (due to network constraints) the model is shared with a coordination server, which will then distribute the changes across all nodes in the network. [21] While this method is inefficient as the infrequent communications mean that many nodes may do much of the same work, and the merging of local models into a global one infrequently may cause loss of information. It still produces a model which converges in a relatively few rounds of communication. [18]

Efforts have been made in techniques to reduce the power memory and storage needed for machine learning algorithms to operate. One of these is Data Stream Mining. The idea is that a device can stream the analytics data directly into the model rather than storing the data in storage for later use. [22] This means after the data has been read by the model it is lost. But that also means that no data needs to be stored, meaning resources are not spent reading and writing to storage. This has an application in mobile devices, as they produce data at a low rate through user interaction. Therefore the model can be build in real time as actions occur. The data produced on the mobile itself may not be enough to effectively train the model, but via communication with other users distributed over the network the model can converge. [18]

From the research available there seems to be research into distributed computing on local devices, investigation in to power measurements of machine learning algorithms [18,21] and power reduction of machine learning algorithms on local devices. [22] But there is a distinct lack of research into efficiency of *distributed algorithms* from the perspective of efficiency and power consumption on local devices. Having a more power efficient distributed machine learning algorithm, even if the optimisation was marginal on each device, would have an enormous effect on the output of the system, as many devices are connected.

3 Problem Description

In the most ideal world, communication between distributed network nodes would not to bottleneck performance. As many nodes as you liked could be added to your network and performance would scale linearly with each node added.

Currently when distributed neural networks become large enough, they become saturated. Adding more nodes no longer increases the speed of computation. To increase the speed of training further by adding more nodes at least one of two things must happen. Either the bandwidth between each node must be increased or the nodes must communicate less while still communicating enough information to continue training. [14] [23] This limit in the rate of computation means that training a neural network cost more time and money. We are at the point now where even if mistakes are found in the largest projects, its often 'infeasible' to retrain the model due to time and monetary cost. [4]

The impact of leaving this problem unsolved will mean the limiting factor of distributed neural networking will be not the processing power of the nodes but the bandwidth between them, bandwidth being one of the scarcest resources in data centres [24]. It will stifle the growth of machine learning models, especially for those that cannot get the funding for high bandwidth compute clusters.

This is why I designed a new paradigm for distributed neural networking that should reduce the communication between each node relative to a generic parameter server. The new paradigm also allows for higher bandwidth between adjacent nodes. As in this scenario nodes need only communicate with their adjacent partner. This results in a distributed learning model that isn't bound by bandwidth when scaling, and should be able to train models to the same level of accuracy in the same amount of time.

4 Implementation

4.1 Tooling

Implementing a distributed neural network is too large a task to be undertaken from scratch. Therefore its necessary to used existing tools, to make the development viable in the time given. This is difficult as not many languages lend themselves to both distributed systems and neural networks.

To ensure high performance the project could be implemented in C++. While C++ is often very performant and also has low level bindings for ML libraries such as TensorFlow. However even the creator of the language sees the need to improve its ability to improve its distributed performance. [25]

Python has great tooling for neural networking, such as TensorFlow [26], and PyTorch [27]. Moreover it has great support for numerical computing with NumPy [28]. These are performant too, by calling C functions or creating code which is optimised to run on GPUs to parallelise computation. However due to the Global Interpreter Lock (GIL) python is infamously bad at concurrency, while its distributed tooling is implemented in native python code, which lack of speed and could bottleneck the performance gained from using NumPy and TensorFlow.

Ultimately I decided to use Elixir as the programming language of implementation. This is because Elixir was designed for developing highly concurrent distributed systems. It does this by having a uniquely brilliant concurrency model. As opposed to OOP languages where 'everything is an object' in Elixir 'everything is a process'. This means the default way of writing the language enables it to be concurrent and scalable. Elixir also has the ability to communicate with other Elixir programs over the network using its own application protocol on top of TCP/IP. Meaning its as easy to communicate with local processes on your own machine as processes on another machine running an Elixir program. Its also been used by artificial intelligence researchers before as the process concurrency models effortlessly lends itself to modelling neurons. [29] Using Elixirs native float and arithmetic implementation would be slower than a C++ or a NumPy implementation, luckily there is a stable package which supports matrix calculations even faster than those in NumPy called Matrex. [30]

The only drawback of using Elixir is at the time of development it didn't have a strong machine learning library, which means implementing the mathematics of the neural network myself. While this was a sizeable amount of work to do, it had the benefit that I didn't to wrestle with an opinionated API such as TensorFlow, I could create my own API to meet my ends.

4.2 Neural Network Implementation

In order to create a distributed neural network. I first needed to create a basic feed forward network that could operate on a single machine. This network doesn't need to be fully featured, its just a means to make an objective comparison between RingTMP and a generic parameter server. Therefore only 2 types of layers were implemented, the hidden layer and the output layer. The hidden layer is a generic dense layer similar to the kind you would find in any other neural network library. The output layer performs similarly to the hidden layer with the key difference that its always the final layer in a network and outputs the activations as probabilities. Within this section I will explain in more detail how the neural network was implemented from scratch and the mathematics behind its function.

4.2.1 Initialisation

Neural networks are composed of layers, while conceptually layers are composed of neurons, they're practically implemented with two components. A weights matrix and a bias vector. As I have already mentioned in this network there are two types of layers, hidden layers and output layers. We need to label each layer with its type, so we know how to perform forward and backpropagation. Therefore we can describe a layer as the tuple:

```
{layer_type, weights, bias}
```

Placing several of these tuples in a list creates a network:

```
network = [{:hidden_layer, weights_1, bias_1},
           {:hidden_layer, weights_2, bias_2},
           {:output_layer, weights_3, bias_3}]
```

The weights and biases have different dimensions depending on the layers input size and output size. A layer might take an activations vector with a size of m and output a size of n. The layer would hold a $n \times m$ matrix and the dimensions of the bias would be $n \times 1$. The output size of one layer must be the input size of the next layer. Initialising the biases is simple, as biases have the function of an intercept in a linear equation, they can be initialised to 0. The trivial code is below:

```
defp initialise_bias(col) do
  Matrex.zeros(col, 1)
end
```

Weights are more complex. Each layer is initialised with random values from a uniform distribution, the shape of the uniform distribution is dependant upon the size of the input and output layers of the neural network. The type of initialisation used is dependant upon the activation function used.

In the hidden layer the ReLU function is used, common wisdom first established in this paper [31] states that for the fastest convergence He initialisation should be used. He initialisation is done by sampling random values from a normal distribution with a mean of 0 and a variance of $2/N$ where N represents the number of input values to a layer.

For the output layer, the softmax function is used. The best initialisation method in this case is using Xavier initialisation. [32] This also takes random samples from a normal distribution with a mean of 0 but has the variance of $1/N$ where N is $(inputSize + outputSize)/2$.

However in practice training the model often failed with He initialisations. This was because of what is known as the 'Dying ReLU Problem'. Which is when the elements of a z vector are negative, the ReLU activation function will return a zero meaning no learning is taking place. Once a neuron becomes dead its unlikely it will be revived as the function is piecewise and provides no slope for recovery such as a Leaky ReLU or a sigmoid function. To remedy this I trialed many distributions, settling on a mean of 0.5 with a variance of 0.25. While this is more simplistic, and may impact training times, its far more likely for the network to not be dead on arrival because of the initialisation parameters. This is part of the code which initialises the matrices in the layers:

```
defp random_val(_x, {n, :he, seed_state, acc}) do
  {val, new_state} = :rand.normal_s(0, 2 / col, seed_state)
  {col, :he, new_state, [val | acc]}
end

defp random_val(_x, {n, :pos, seed_state, acc}) do
  {val, new_state} = :rand.normal_s(0.5, 0.25, seed_state)
  {n, :pos, new_state, [val | acc]}
end

defp random_val(_x, {n, :xavier, seed_state, acc}) do
  {val, new_state} = :rand.normal_s(0, 1 / col, seed_state)
  {col, :xavier, new_state, [val | acc]}
end
```

You can find the wider context for this code snippet in code listing in the Appendix 1. 1

4.2.2 Forward Propagation

In forward propagation we give an input vector and an output vector is returned. In order for that to happen the input vector is passed through each layer, each time being transformed by the weights, bias and activation function of that layer. The input to the network could be measurements describing a flower (like in the setosa dataset), the output layer could describe which species. More broadly put, the input described the features, and the output predicts the categories to which those features belong.

On a layer level forward propagation is performed by taking the input vector multiplying it with the weights matrix, after which you add the bias and apply the activation function. The output of this is then passed to the next layer, or if its the last layer, output as the network prediction result. The generic mathematics of a forward function is described below in this linear equation where X is the input matrix, W is the weights, B is the bias, $activationFunc(x)$ is the activation function and finally A is the output activation:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad W = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \qquad B = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{4}$$

$$A = activationFunc(W^T X + B)$$

In my implementation there are only two layer types. Hidden layers and output layers. The only difference between these two layers is the activation function that they use. Hidden layers use the ReLU activation function which is a simple piecewise non-linear function described as such:

$$relu(z) = max(0, z) \tag{5}$$

This function has become the de facto application function in dense hidden layers since its debut in 2011. [33]. In Elixir the forward action in the hidden layer is implemented like so: [1]

```
defmodule HiddenLayer
  def forward(previous_activation, weights, bias) do
    weights |> Matrex.transpose()
    |> Matrex.dot(previous_activation)
    |> Matrex.add(bias)
    |> relu()
  end

  defp relu(z_vector) do
    z_vector|> Matrex.apply(
```

[1]The pipe operator `|>` transforms the function `val_a |> a_function(val_b)` into `a_function(val_a, val_b)`

```
      fn value, _index -> if value > 0, do: value, else: 0 end
  )
 end

 ...
```

The output layer uses a more complex activation function, the softmax function, which transforms its inputs into probabilities. The sum of these probabilities is always 1. This is the softmax function, where z is a $i \times 1$ vector:

$$softmax(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \tag{6}$$

This function is implemented the categorical output layer like so:

```
defmodule CategoricalOutputLayer do
 def forward(previous_activation, weights, bias) do
    weights |> Matrex.transpose()
    |> Matrex.dot(previous_activation)
    |> Matrex.add(bias)
    |> softmax()
 end

 defp softmax(z_vector) do
    stabilised_vec = Matrex.subtract(z_vector, Matrex.max(z_vector))
    exp = Matrex.apply(stabilised_vec, :exp)
    Matrex.divide(exp, Matrex.sum(exp))
 end

 ...
```

Propagating forward through the layers happens by using the output of one layer as the input to the next, performing a slightly different forward action depending on the layer:

PUT IN APPENDIX

```
defmodule FeedForwardNetwork.Forward do
 def forward(network, input_vector) do
   Enum.reduce(network, input_vector, fn layer, acc ->
       forward_layer(layer, acc) end)
 end

 defp forward_layer({:hidden_layer, weights, bias}, prev_activation) do
   {_z_vec, activation} = HiddenLayer.forward(prev_activation, weights,
       bias, [])
   activation
 end

 defp forward_layer({:output_layer, weights, bias}, prev_activation) do
   CategoricalOutputLayer.forward(prev_activation, weights, bias, [])
 end

 ...
```

4.2.3 Cost Function

A neural network is trained by using examples of correct classifications based on given data. The data is input into the network, and the network attempts to label the data correctly. The networks prediction is compared to the correct classification using a cost function, this gives a single value representation describing how close or far the network was to the correct classification. A perfect classification will result in a score of 0, the more wrong the network prediction is the larger the cost produced by the cost function. The cost function used in this network is categorical cross entropy, which allows for multi class classification. It takes two inputs, an output vector y of the network (its prediction), and the one hot encoded vector t of the target classification:

$$cost(t, y) = -\sum_{i}^{C} t_i log(y_i) \tag{7}$$

This implementation is similar, but removed redundant computation in the cases where $t_i = 0$ the cost will always be 0. Here we also catch situations where the activation underflows to 0, which would result in a math error as $log(0)$ is undefined. For the full code example see the code listing 2 in the appendix.

```
defmodule CategoricalOutputLayer
  def loss(activation, target_activation) do
    target_position = get_target_category(target_activation)

    activation
    |> Matrex.at(target_position, 1)
    |> (fn x ->
         if x == 0.0, do: 100, else: -:math.log(x) end
       end).()
  end

  ...
```

4.2.4 Back Propagation

The result of the cost function represents the error within the network, backpropagation adjusts the weights and biases to minimise the loss within the network. Meaning that if the same input was given to the network again after backpropagation, the output would be closer to the target, than before backpropagation had occurred. The changes to the weights and bias of each layer can be calculated by using the chain rule, to discover the relationship between the weights and biases with respect to the loss.

To calculate the change in weights and biases in the output layer, we first need to find the derivative of the cost function w.r.t the network output and the derivative of the softmax function w.r.t z.

$$\begin{aligned} c &= -\sum_i^c t_i log(a_i) \\ \frac{\partial c}{\partial a} &= -\sum_i^c \frac{t_i}{a_i} \end{aligned} \tag{8}$$

Finding the derivative of the softmax is a little more complex:

$$\begin{aligned} & & a_j &= \frac{e^{z_j}}{\sum_{k=1}^N e^{z_k}} \\ &\text{applying quotient rule (for brevity} \sum = \sum_{k=1}^N e^{z_k}) & \frac{\partial a_i}{\partial z_j} &= \frac{e^{z_i}\sum - e^{z_j}e^{z_i}}{\sum^2} \\ &\text{if } i = j \text{ then it can be factorised} & \frac{\partial a_i}{\partial z_j} &= \frac{e^{z_i}}{\sum}\frac{\sum - e^{z_j}}{\sum} \\ &\text{substituting in softmax} & \frac{\partial a_i}{\partial z_j} &= a_i(1-a_j) \\ &\text{if } i \neq j \text{ then it can be factorised} & \frac{\partial a_i}{\partial z_j} &= \frac{0 - e^{z_j}e^{z_i}}{\sum^2} \\ &\text{substituting in softmax} & \frac{\partial a_i}{\partial z_j} &= -a_j a_i \end{aligned} \tag{9}$$

Combining these with the chain rule ($\frac{\partial c}{\partial z} = \frac{\partial c}{\partial a} \cdot \frac{\partial a}{\partial z}$):

$$\begin{aligned} \frac{\partial c}{\partial z_i} &= \sum_k^C \frac{\partial c}{\partial a_k}\frac{a_k}{z_i} \\ &= \frac{\partial c}{\partial a_i}\frac{a_i}{z_i} - \sum_{k\neq i}\frac{\partial c}{\partial a_k}\frac{a_k}{z_i} \\ &= -t_i(1-a_i) + \sum_{k\neq i} t_k a_i \\ &= -t_i + a_i \sum_k t_k \\ &= a_i - t_i \end{aligned} \tag{10}$$

The rest of the calculations will be in linear equations as its closer to the implementation as easier to follow, where:

- T is a $m \times 1$ target output
- A is a $m \times 1$ output activation
- W is a $n \times m$ weights matrix
- B is a $m \times 1$ bias
- Z is a $m \times 1$ sum of the matrix multiplication and bias
- A^{L-1} is a $n \times 1$ activation of the previous layer

Output Layer bias w.r.t cost:

$$\begin{aligned}
Z &= W^T A_{L-1} + B \\
\frac{\partial Z}{\partial B} &= 1 \\
\\
\frac{\partial c}{\partial B} &= \frac{\partial c}{\partial Z} \cdot \frac{\partial Z}{\partial B} \\
\frac{\partial c}{\partial B} &= A - T \cdot 1
\end{aligned} \tag{11}$$

Output Layer weights w.r.t cost:

$$\begin{aligned}
Z &= W^T A_{L-1} + B \\
\frac{\partial Z}{\partial W} &= A_{L-1} \\
\\
\frac{\partial c}{\partial W} &= \frac{\partial Z}{\partial W} \cdot \frac{\partial c}{\partial Z} \\
\frac{\partial c}{\partial W} &= A_{L-1} \cdot (A - T)^T
\end{aligned} \tag{12}$$

Output layer remaining error:

$$\begin{aligned}
Z &= W^T A_{L-1} + B \\
\frac{\partial Z}{\partial A_{L-1}} &= W \\
\\
\frac{\partial c}{\partial A_{L-1}} &= \frac{\partial Z}{\partial A_{L-1}} \cdot \frac{\partial c}{\partial Z} \\
\frac{\partial c}{\partial A_{L-1}} &= W \cdot (A - T)
\end{aligned} \tag{13}$$

We use the remaining error from the output layer to calculate the weights and biases w.r.t the cost in the hidden layers:

- $\frac{\partial c}{\partial A_{L-1}}$ is a $n \times 1$ remaining error vector
- A_{L-1} is a $n \times 1$ output activation of this layer
- W is a $l \times n$ weights matrix of this layer
- B is a $n \times 1$ bias of this layer
- Z is a $n \times 1$ sum of the matrix multiplication and bias
- A^{L-2} is a $l \times 1$ activation of the previous layer

The differential of the ReLU function.

$$\begin{aligned} A_{L-1} &= max(0, Z) \\ \\ \frac{\partial A_{L-1}}{\partial Z} &= \begin{cases} z_i > 0 & 1 \\ else & 0 \end{cases} \end{aligned} \tag{14}$$

This can be combined with $\frac{\partial c}{\partial A_{L-1}}$ to make $\frac{\partial c}{\partial Z}$ ($\odot$ denotes element-wise multiplication):

$$\frac{\partial c}{\partial Z} = \frac{\partial c}{\partial A_{L-1}} \odot \frac{\partial A_{L-1}}{\partial Z} \tag{15}$$

Hidden Layer bias w.r.t cost:

$$\begin{aligned} Z &= W^T A_{L-2} + B \\ \frac{\partial Z}{\partial B} &= 1 \\ \\ \frac{\partial c}{\partial B} &= \frac{\partial c}{\partial Z} \cdot \frac{\partial Z}{\partial B} \\ &= \frac{\partial c}{\partial Z} \cdot 1 \end{aligned} \tag{16}$$

Hidden Layer weights w.r.t cost:

$$\begin{aligned}
Z &= W^T A_{L-2} + B \\
\frac{\partial Z}{\partial W} &= A_{L-2} \\
\\
\frac{\partial c}{\partial W} &= \frac{\partial Z}{\partial W} \cdot \frac{\partial c}{\partial Z} \\
&= A_{L-2} \cdot \frac{\partial c}{\partial Z}
\end{aligned} \tag{17}$$

Hidden Layer error w.r.t cost:

$$\begin{aligned}
Z &= W^T A_{L-2} + B \\
\frac{\partial Z}{\partial A_{L-2}} &= W \\
\\
\frac{\partial c}{\partial A_{L-2}} &= \frac{\partial c}{\partial Z} \cdot \frac{\partial Z}{\partial A_{L-2}} \\
&= \frac{\partial c}{\partial Z} \cdot W
\end{aligned} \tag{18}$$

We use these derivatives to change the values of the weights and biases, however using them on their own creates too much change in the network causing it to take too large steps, often resulting in the better solutions being missed. Therefore we apply a learning rate, a scalar value which forces the model to take smaller steps. We use the learning rate like so:

$$\begin{aligned}
\Delta B &= \frac{\partial c}{\partial B} \cdot \text{learning rate} \\
\Delta W &= \frac{\partial c}{\partial W} \cdot \text{learning rate}
\end{aligned} \tag{19}$$

In my code you can see that I use the exact same equations as in the mathematics. In the output layer:

```
defmodule CategoricalOutputLayer do

  def back(activation, prev_activation, weights, bias, target_activation,
      opts \\ []) do
    cost_wrt_z = Matrex.subtract(target_activation, activation)

    weight_delta =
      Matrex.dot_nt(prev_activation, cost_wrt_z)
      |> Matrex.multiply(Keyword.get(opts, :learning_rate, 1.0))

    bias_delta = cost_wrt_z |> Matrex.multiply(Keyword.get(opts,
        :learning_rate, 1.0))

    remaining_error = Matrex.dot(weights, cost_wrt_z)

    new_weights = Matrex.add(weights, weight_delta, 1, 1)
    new_bias = Matrex.add(bias, bias_delta)
    {new_weights, new_bias, remaining_error}
  end
  ...
```

And in the hidden layer:

```
defmodule HiddenLayer do

 def back(z_vec, prev_activation, weights, bias, remaining_error, opts \\
    []) do
  learning_rate = Keyword.get(opts, :learning_rate, 1.0)
  a_wrt_z = Matrex.apply(z_vec, fn value, _index -> if value > 0, do: 1,
     else: 0 end)
  cost_wrt_z = Matrex.multiply(remaining_error, a_wrt_z)

  weight_delta =
  prev_activation
  |> Matrex.dot_nt(cost_wrt_z)
  |> Matrex.multiply(learning_rate)

  bias_delta = cost_wrt_z |> Matrex.multiply(learning_rate)

  remaining_error = Matrex.dot(weights, cost_wrt_z)
  new_weights = weights |> Matrex.add(weight_delta, 1, 1)
  new_bias = Matrex.add(bias, bias_delta)

  {new_weights, new_bias, remaining_error}
 end
 ...
```

4.3 Parameter Server

The Parameter Server is currently the most popular way to distribute a machine learning task. Therefore I need to create one in order to compare it with my solution. My parameter server functions by iteratively sending the batches of training pairs with the parameters then when all the workers have responded the parameters from each worker are averaged. This cycle repeats until the network is trained.

Connecting the workers to the server is quite simple in Elixir, here is an example using the elixir language shell, iex:

```
Machine 1
$ iex --name foo@192.168.1.11 --cookie shared_string

Machine 2
$ iex --name bar@192.168.1.12 --cookie shared_string
iex(bar@192.168.1.12)> Node.ping(:"foo@192.168.1.11")
:pong
```

In elixir there is a concept of nodes which each node has its own name in the format of `<application>@<local_ip>` you can have multiple nodes on a single machine, but if you need to communicate with nodes on another machine, you must set a cookie and ensure the cookie is the same on both the machines. In the example above we see one node messaging another one to get a response.

We can also use this connection to send commands to a process within another node, have it do work for us and then respond with the result:

```
def send_to_worker(
   sup, {train_input, train_label}}, network, opts) do

  # sup = {TaskSupervisor, node} pair

  sup
  |> Task.Supervisor.async(
    FeedForwardNetwork.Back,
    :back_once,
    [train_inputs, train_labels, network, opts]
  )
end
```

The `TaskSupervisor` is a named process, the `node` is a variable holding the atom describing the node e.g. (`workerone@192.168.1.14`) this indicates the specific process you want to send the task to. Within the async function the model, function and function arguments are expressed. This asynchronously returns a task struct which can be `Task.await(task)` awaited upon. This means a batch can be sent to each worker node at the same time and awaited on for the result. Like in the code below:

```
  def send_to_workers(chunk, network, opts \\ []) do
    chunk
    |> Enum.map(
      fn sup_train_pair ->
        send_to_worker(sup_train_pair, network, opts)
      end)
    |> Enum.map(fn t -> Task.await(t) end)
  end
```

Each chunk contains elements destined for each node. These are asynchronously sent and awaited upon until all of them have sent back their updated versions of the network. These are then averaged and become the new parameters of the parameter server.

```
  def avg_networks(networks) do
    size = Enum.count(networks)

    networks
    |> Enum.reduce(
      fn network, sum_acc ->
        sum_network(network, sum_acc)
      end)
    |> divide_network(size)
  end
```

For the complete code look in the code appendix .... ¡PUT IN CODE APPENDIX¿

4.4 RingTMP

The main product of my project is the RingTMP network, the name is an acronym for *Ring Topological Model Parallel.* Its called as such because it has a ring topology and the neural network model is split between the nodes. In RingTMP each node in the distributed network holds consecutive layers of the neural network, there are two types of node, the master node and the worker nodes. The master node holds the training and testing data. It also starts the forward propagation and back propagation process, while worker nodes receive messages from adjacent nodes which they process and then forward the message to the next node.

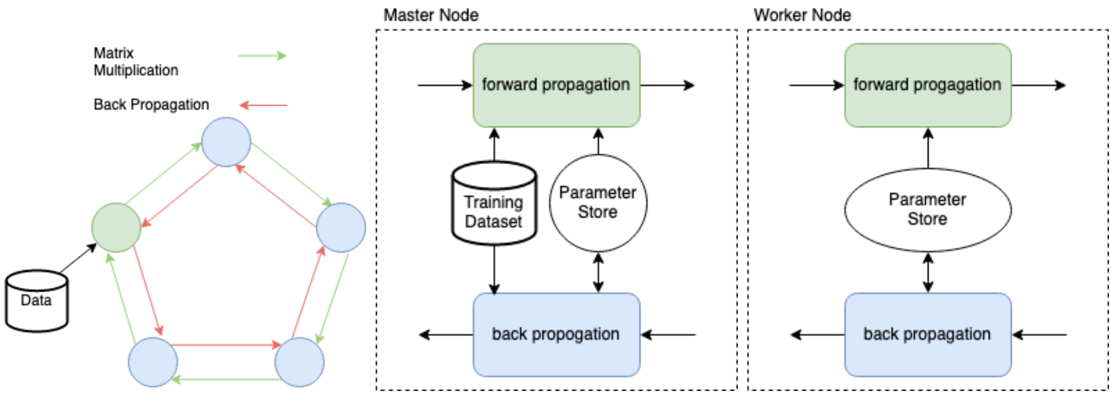


Figure 2: Left: An example network of RingTMP. The green node being the master node and the blue nodes being the worker nodes. Centre: The architecture of the master node. Right: The architecture of the worker node.

Each batch of data follows the same path. It starts in the master node, a batch is forward propagated through the segment of the neural network being held locally. The intermediate result of that is send to the first worker node in the chain. Which uses the intermediate values to forward propagated through its neural network layers and send its results to the next node in the RingTMP network. This continues until the last worker node in the chain makes a prediction about the data given. This is then sent back to the master node which calculates the loss and sends that back to the worker. The worker then uses this to begin the backpropagation of errors back through the network. It first updates its own parameters based on the loss. Then passes back the remaining error back to the node before it until all the parameters have been updated.

This is different to the parameter server in several important ways. First the model is split between the nodes, this means all thing being equal a larger model can be trained on a RingTMP network than on a non-model parallel parameter server. This is more of a useful feature for embedded devices where memory is at a premium. Using the example of the EfficientNetV2-L model, which contains 121 million parameters. [34] Estimating each of those to be a double precision value it would amount to almost a 1 Gigabyte of data, excluding the intermediate values stored in the RAM while performing calculations. A non-model parallel parameter server has to be able to hold all the variables in order for it to function, whereas in a RingTMP network the parameters can be distributed between the layers.

Another major advantage is decreased communication between nodes. A non-model parallel parameter server will always send more data for a layer of the same size. This can be shown to be true thus.

The amount of arbitrary units of data sent from a parameter server to a worker where n is the input size and m is the output size. Where $m \geq 1$ and $n \geq 1$ the this can be shown to be:

$$\begin{aligned} f(n,m) &= (n \cdot m) + m \\ f(n,m) &= m(n+1) \end{aligned} \tag{20}$$

The amount of arbitrary units of data sent from a RingTMP node to another node where m is the input size and n is the output size this can be shown to be:

$$g(n,m) = 2m \tag{21}$$

substituting $g(n,m)$ into $f(n,m)$:

$$\begin{aligned} f(n,m) &= \frac{g(n,m)}{2} \cdot (n+1) \\ f(n,m) &= g(n,m) \cdot \frac{n+1}{2} \end{aligned} \tag{22}$$

Given that n must be at least 1, $\frac{n+1}{2}$ must also be at least 1. A number multiplied by 1 or greater will always be greater than or equal to the initial value, we can say that the parameter server will always communicate as much or more information between nodes than RingTMP.

A simple example of this can be given to show the reduction in communication between nodes. Take a two layer neural network the first layer having 784 input nodes and 50 output nodes, and the second layer had 50 input nodes and 10 output nodes. A worker sending updated weights back to a parameter server would have to communicate 39760 values $((784 \times 50) + (50 \times 10) = 39860)$. Whereas for the RingTMP model the same neural network split over two nodes would only need to communicate 120 values. Over 300 times less network communication.

The final major advantage is the capacity for increased bandwidth between the nodes, which is built into the structure of the network. In a parameter server model all workers connect to a single parameter server, so the network have a star topology. This means bandwidth of the network is only as large as the bandwidth of the parameter server, the relationship between the parameter server bandwidth with each worker and the number of nodes in inversely proportional ($b = \frac{1}{n-1}$ where $n \geq 2$). With the RingTmp network all nodes have two connections, one ahead, one behind in a ring topology. This means no single node will limit the bandwidth of the network. Meaning that scaling the network will have no impact to any nodes bandwidth if the network is physically arranged in a ring topology.

I implemented RingTMP in Elixir with the help of a GenServer, as the name suggests its a structure that acts like a server holding state and sending and receiving messages. Elixir also treats the GenServer like a server, separating out the client and implementation code.

First each node must be initialised, it takes its the `node_type`, this is either `:master | :worker` and determines the behaviour of the node. `m_node, this_node, prev_node, next_node` are all tuples which contain { process, node } tuples which point to the `master, current, previous, next` nodes in the network respectively. The current node is then started and the initial state is set.

```
@impl true
def init({node_type, m_node, this_node, prev_node, next_node, nn_opts}) do

  this_host = elem(this_node, 1)
  Node.start(this_host, :shortnames)

  {:ok, {node_type, m_node, this_node, prev_node, next_node, %{}, %{},
      %{accuracy: 0, cost: 0}, 0, false, nn_opts}}
end
```

Implementing the forward propagation code in the master node looks like so:

```
  # client code
  def train(batches, expecteds) do
     GenServer.call(__MODULE__, {:train, batches, expecteds})
  end

  # callback
  @impl true
  def handle_call(
    {:train, all_batches, all_expecteds}, _from,
    state = {:master, _m_node, _this, prev_node, next_node, _history,
        _expecteds, test_acc, max_bid, false, _nn_opts}
  ) do

    # sends batches around the network
    Enum.map(all_batches, fn batch ->
      send(next_node, {:forward, forward_batch(batch)})
    end)

    # master needs to store the expected values to calculate loss
    expecteds = Enum.reduce(
      all_expecteds, %{},
      fn expected, acc -> add_history(acc, expected) end)

    # also needs to store sent batches in order to
    # update its own neural network parameters
    history = Enum.reduce(
      all_batches, %{},
      fn batch, acc -> add_history(acc, batch) end)

    # stores batch id so we know when last element is finished
    max_bid = elem(List.last(all_batches), 0)

    state = state
    |> put_elem(5, history)
    |> put_elem(6, expecteds)
    |> put_elem(8, max_bid)
    |> put_elem(9, true) # making master busy

    {:reply, :ok, state}
  end

  defp forward_batch({num, batch_data}) do
    {num, NeuralNet.forward(batch_data)}
  end
```

you may notice the use of batches. A batch is training data linked with a sequential ID, this is so we can track batches (`{batch_id, data}`) through the network, the expected values (or labels for the data) have a corresponding ID so they can be matched to calculate loss and accuracy while testing and training. You can also see here the state is updated with the history, the input activations to the RingTmp node. We need this as an input parameter for the backpropagation function. The expecteds only the master node needs to hold, its a map of all the labels the corresponds to a piece of training data. We store the ID of the last training pair so we know when an epoch has finished. Finally the `true` value sets the state of the network to busy so you can't accidentally train the network on two different datasets.

A worker receives `{:forward, batch}` and continues the forward propagation for every batch it receives.

```
@impl true
def handle_info({:forward, batch}, state = {:worker, _m_node, _this,
    _prev, next_node, history, _exp, _test_acc, _max_bid, busy,
    _nn_opts}) do
  # do forward action
  {num, _} = batch

  send(next_node, {:forward, forward_batch(batch)})

  history = add_history(history, batch)
  state = put_elem(state, 5, history)
  {:noreply, state}

end
```

When a worker node sends its activations to the master, the master node matches the output activation to the expected label using the batch ID, the loss is then calculated and a message is sent back to the same worker `{:back, {b_id, loss}}` .

```
@impl true
def handle_info({:forward, batch}, state = {:master, _m_node, _this,
    prev_node, _next_node, _history, expecteds, _test_acc, _max_bid,
    busy, _nn_opts}) do

  # do back
  {b_id, b_data} = batch
  id_as_str = Integer.to_string(b_id)
  expected = Map.get(expecteds, id_as_str)

  send(prev_node, {:back, {b_id, expected}})
  {:noreply, state}
end
```

This worker uses the error to update its own parameters, then sends the error it produces back further.

```
@impl true
def handle_info({:back, error_batch}, state = {:worker, _m_node, _this,
    prev_node, _next, history, _exp, _test_acc, _max_bid, _busy,
    nn_opts}) do

  {b_id, error_vec} = error_batch
  b_id_str = b_id |> Integer.to_string()
  input_for_batch = Map.get(history, b_id_str)
  remaining_error = NeuralNet.back(input_for_batch, error_vec, nn_opts)

  send(prev_node, {:back, {b_id, remaining_error}})

  {:noreply, state}
end
```

When this finally reaches the master node again is updates the parameters in its segment of the network, then changes the state of the RingTmp network to not busy if its processed the last batch.

```
@impl true
def handle_info({:back, error_batch}, state = {:master, _m_node, _this,
    _prev, _next, history, _exp, _test_acc, max_bid, _busy, nn_opts}) do

  {b_id, error_vec} = error_batch
  b_id_str = b_id |> Integer.to_string()
  input_for_batch = Map.get(history, b_id_str)
  remaining_error = NeuralNet.back(input_for_batch, error_vec, nn_opts)

  if b_id == max_bid do
    state = put_elem(state, 9, false)
    {:noreply, state}
  else
    {:noreply, state}
  end
end
```

5 Results

In the introduction I stated these aims:

- To create a prototype RingTMP framework.
- To create a parameter server model framework.
- To demonstrate less communication between nodes
- To demonstrate that RingTMP is at least as scalable than a generic parameter server
- To demonstrate RingTMP can train neural networks to at least the same accuracy in at least the same amount of time as an equivalent parameter server

As can be seen in the implementation section the first two aims have been completed. Now using experimental results it will be shown to what degree the other aims were completed.

For experimental results we needed datasets that we could use to train the neural networks. The first dataset chosen was the iris dataset, this is a dataset which describes the the length and width of sepals and petals from 3 different species of flower. This dataset was chosen because of its small feature dimension and limited examples (150), to see how well the different network types would be effected by small feature dimension.

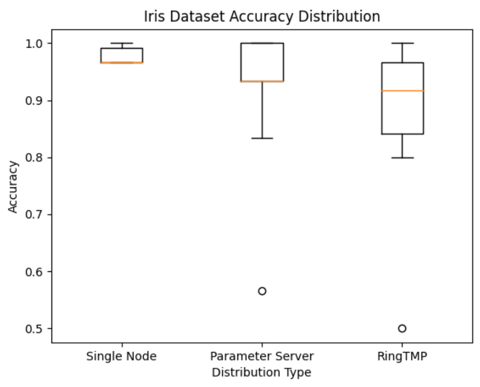
The second dataset used was the MNIST numbers dataset. [35] This second dataset contrasts the first. Firstly this is because it has a comparatively greater feature input of 784 (a flattened 28 by 28 image). Secondly it a large testing and training set of 60000 training example and 10000 test examples.

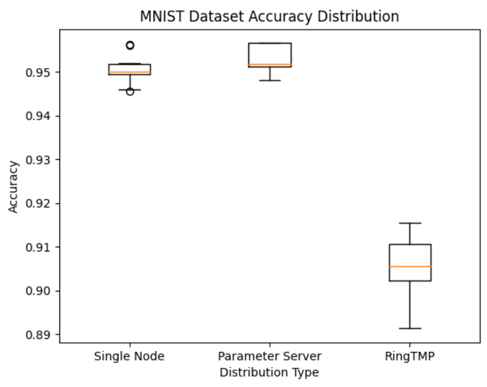
Several experiments were carried out. The first experiment was training models using the Iris dataset, this experiment was repeated 10 times for reliability and was performed on the single node implementation of the neural network, the parameter server implementation and the RingTMP implementation. Each implementation used the same neural network model, consisting of 4 layers of node numbers 4,8,8,3. The parameter server was configured with 1 server and 2 workers (3 nodes in total) and RingTMP was configured with 3 nodes 1 master 2 workers. Both these are using the same number of nodes and should represent comparative performance. Every test session ran for at most 100 epochs, or until the test result failed to improve for 3 epochs.

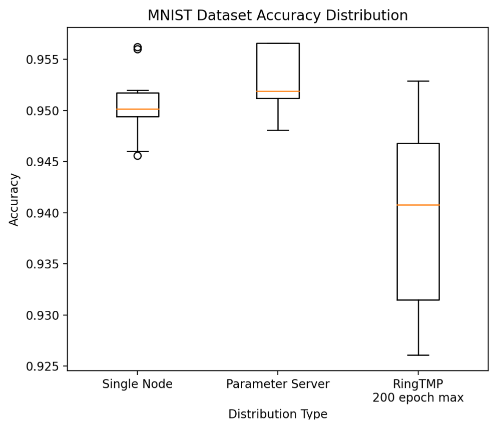
The second experiment was training models using the MNIST numbers dataset, this experiment was repeated 5 times and was also performed on each implementation. The neural network used in each implementation had 4 layers with a 784,50,50,10 node configuration. As before RingTMP was configured with 3 nodes as was the parameter server. Every test session ran for at most 20 epochs, or until the test result failed to improve for 3 epochs. But later to better compare performance the RingTMP tests were run again with a maximum of 200 epochs.

More experiments were also performed which compared the scalability of the parameter server with RingTMP. In these tests a 10 layer deep neural network model was used. This has to be done as RingTMP is a model parallel framework, it can only scale to as many layers the neural network has. Each test is repeated 5 times and changes the amount of nodes each framework uses, to see whether its speed increases or decreases over time.

5.1 Accuracy

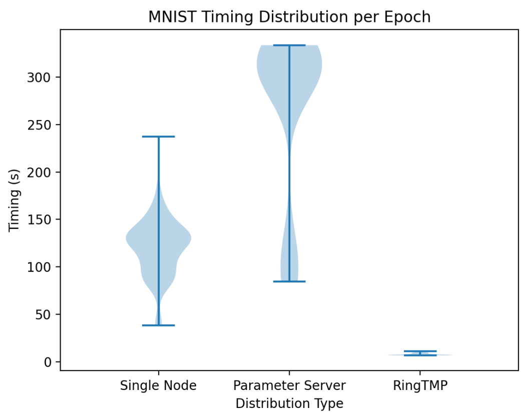


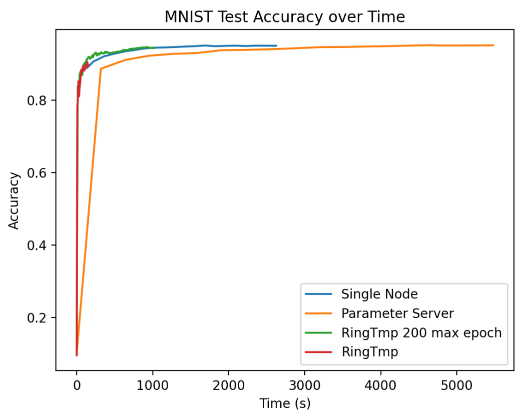




Here we see a box plot with the distribution of highest accuracies of each training session for both the Iris and MNIST datasets. All distribution types performed well on the Iris dataset each reaching a perfect score. Applying a two tailed hypothesis Mann-Whitney U test where $\alpha = .05$ shows that there isn't enough statistical significant to show they are sampled from a different population. Whereas its clear in the MNIST testing that RingTMP's accuracy was not as high as the others. Applying the Mann-Whitney U test again shows that Single Node and Parameter Server implementation are likely to have the same performance distribution. RingTMP's accuracy was much lower than the other two approaches. This was because it only took a short amount of time to complete an epoch, but had a slower learning rate per epoch. The other two distributions types performance often peaked before the 20th epoch as RingTMP was still learning. To fully explore RingTMP accuracy performance, the MNIST test for RingTMP were performed again this time with a 200 epoch limit. As you can see from above this shows that RingTMP can achieve similar results to the single node and parameter server implementations, albeit less frequently.

5.2 Timings

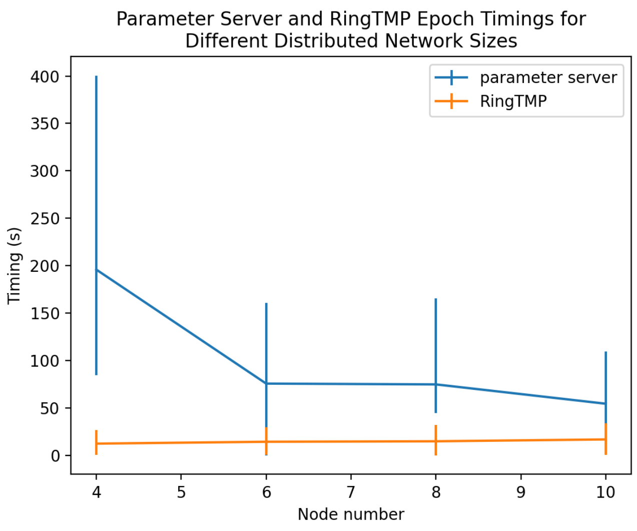




From this violin plot observe how quickly and consistently RingTMP can iterate through each epoch (always between 7-12 seconds), this could be in part due to the higher throughput and concurrent nature of the RingTMP network however it must also be recognised that this may also be due to a bug. While also notice the top heavy distribution of the parameter servers timings, more often than not taking over 250 seconds for a single epoch, even slower than the single node timings. Which I suspect is due to the transmission of parameters to nodes and the time taken to aggregate parameters by the parameter server. However while RingTMP epochs are faster, its seems to learn less from each epoch.

In the second chart of the figure we can see that RingTMP, while its learning is a little more irratic seems to learn at the same rate as a single node, when given a higher epoch limit. We can also see how much we were stunting RingTMP's performance by limiting it to only 20 epochs, observe the red line being cut short only 150 seconds into training. We can also see the parameter servers slow rate of learning relative to the other two methods, although eventually it achieved a higher accuracy than the other two.

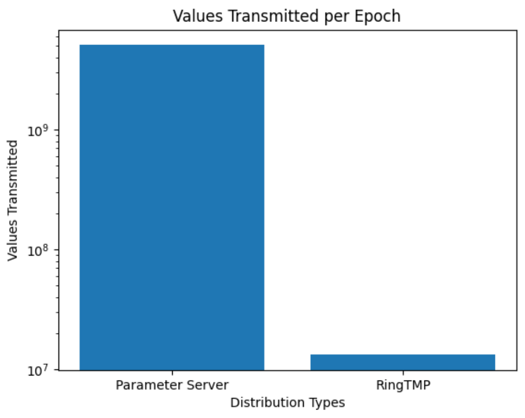
5.3 Scalability

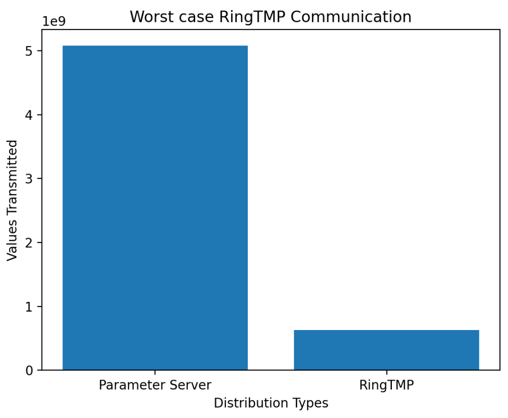


To be a good distributed model, a model must be able to handle many nodes without performance decreasing. However adding and removing nodes in each of these models should have a different effect on the timing of epochs. We can see that the parameter server shows a inversely proportional relationship between the timing of an epoch and the number of nodes. However the results for the RingTMP show a flat line. What would be expected is that the more more nodes the model has the faster it would train, while the computation of the nodes is the limiting factor. You could read the results of the RingTMP this way, although to be sure you would need more evidence of the time waiting in comparison to processing.

5.4 Communication

On of the key goals of RingTMP was to reduce communication between nodes while running the tests the communication of parameters and activations were being monitored. On the right figure we can see that RingTMP has clearly achieved its key goal of reducing communication between the nodes. Using the MNIST network of 4 layers and





784,50,50,10 node configuration it was shown that RingTMP communicates 1.32×10^7 values each epoch. However the parameter server communicates 5.0772×10^9 each epoch, two orders of magnitude more. However we know that RingTMP epochs are far faster than parameter server epochs. To make a comparison about values communicated per unit time we can take the fastest RingTMP epoch (7 seconds) and the slowest parameter server epoch (333.5 seconds). In this case for every parameter server epoch 47.643 (3.dp) RingTMP epochs occur. Even in this most extreme situation the RingTMP network still communicates less, as can be seen from the above figure on the left.

5.5 Summary

In summary RingTMP is nearly as accurate as the Parameter Server and the Single Node neural network. It performed similarly to the others in the Iris dataset tests, and can achieve similar results once given the same amount of time to train as the others. Still its clearly not as accurate as the other implementations so this aim was not realised. However while its accuracy is not as high as the parameter server, its rate of training, especially initially is faster.

However there is not enough evidence to support that RingTMP is as scalable as the parameter server, seeing as adding more nodes to RingTMP didn't produce a reduction in time per epoch.

Most importantly its clear to see that the RingTMP framework exhibits much less communication in comparison to a parameter server of a similar size, with the difference being in orders of magnitude.

References

[1] WIRED UK Matt Burgess. Some uk stores are using facial recognition to track shoppers.

[2] Scott Mayer McKinney, Marcin Sieniek, Varun Godbole, Jonathan Godwin, Natasha Antropova, Hutan Ashrafian, Trevor Back, Mary Chesus, Greg S. Corrado, Ara Darzi, Mozziyar Etemadi, Florencia Garcia-Vicente, Fiona J. Gilbert, Mark Halling-Brown, Demis Hassabis, Sunny Jansen, Alan Karthikesalingam, Christopher J. Kelly, Dominic King, Joseph R. Ledsam, David Melnick, Hormuz Mostofi, Lily Peng, Joshua Jay Reicher, Bernardino Romera-Paredes, Richard Sidebottom, Mustafa Suleyman, Daniel Tse, Kenneth C. Young, Jeffrey De Fauw, and Shravya Shetty. International evaluation of an ai system for breast cancer screening. *Nature*, 577(7788):89–94, Jan 2020.

[3] AJ Piergiovanni and Michael S. Ryoo. Avid dataset: Anonymized videos from diverse countries, 2020.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[5] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020.

[6] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, October 2014. USENIX Association.

[7] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks, 2018.

[8] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing. Strads: A distributed framework for scheduled model parallel machine

learning. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[9] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012.

[10] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[11] David Newman, Padhraic Smyth, Max Welling, and Arthur U Asuncion. Distributed inference for latent dirichlet allocation. In *Advances in neural information processing systems*, pages 1081–1088, 2008.

[12] Alexander Smola and Shravan Narayanamurthy. An architecture for parallel topic models. *Proc. VLDB Endow.*, 3(1–2):703–710, September 2010.

[13] Amr Ahmed, Moahmed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J. Smola. Scalable inference in latent variable models. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, WSDM '12, page 123–132, New York, NY, USA, 2012. Association for Computing Machinery.

[14] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems*, pages 19–27, 2014.

[15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[16] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data*, 1(2):49–67, 2015.

[17] Anis Elgabli, Jihong Park, Amrit S Bedi, Mehdi Bennis, and Vaneet Aggarwal. Gadmm: Fast and communication efficient framework for distributed machine learning. *Journal of Machine Learning Research*, 21(76):1–39, 2020.

[18] Jakub Konečnỳ, H Brendan McMahan, Daniel Ramage, and Peter Richtárik. Federated optimization: Distributed machine learning for on-device intelligence. *arXiv preprint arXiv:1610.02527*, 2016.

[19] Chenxin Ma, Jakub Konečný, Martin Jaggi, Virginia Smith, Michael I. Jordan, Peter Richtárik, and Martin Takáč. Distributed optimization with arbitrary local solvers. *Optimization Methods and Software*, 32(4):813–848, 2017.

[20] M. Chiang and T. Zhang. Fog and iot: An overview of research opportunities. *IEEE Internet of Things Journal*, 3(6):854–864, 2016.

[21] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan. When edge meets learning: Adaptive control for resource-constrained distributed machine learning. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 63–71, 2018.

[22] Eva García-Martín, Crefeda Faviola Rodrigues, Graham Riley, and Håkan Grahn. Estimation of energy consumption in machine learning. *Journal of Parallel and Distributed Computing*, 134:75 – 88, 2019.

[23] P. Sun, Y. Wen, T. N. Binh Duong, and S. Yan. Timed dataflow: Reducing communication overhead for distributed machine learning systems. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1110–1117, 2016.

[24] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition.* 2013.

[25] *Ask Me Anything with Bjarne Stroustrup, hosted by John Regehr.* YouTube, Jul 2020.

[26] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

[28] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe,

Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with numpy. *Nature*, 585(7825):357–362, Sep 2020.

[29] Gene Sher. *Handbook of Neuroevolution Through Erlang.* 11 2012.

[30] Stas Versilov. Matrex. https://hexdocs.pm/matrex/Matrex.html.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[32] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[33] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323. JMLR Workshop and Conference Proceedings, 2011.

[34] Mingxing Tan and Quoc V Le. Efficientnetv2: Smaller models and faster training. *arXiv preprint arXiv:2104.00298*, 2021.

[35] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

Appendices

A Appendix 1

A.0.1 Code Listings

Listing 1: Network Initialisation

```
defmodule FeedForwardNetwork.DefineNetwork
  defp initialise_parameters(definition, initial_seed) do
    definition
    |> Enum.reduce(
      {[], initial_seed},
      fn layer, {acc, seed_state} ->
        {layer, new_state} = initialise_layer(layer, seed_state)
        {acc ++ [layer], new_state}
      end
    )
  end

  defp initialise_layer({layer_type, input_size, output_size}, seed_state)
      do
    init_type = layer_to_init(layer_type)

    {initial_weights, seed_state} =
      initialise_weights(input_size, output_size, init_type, seed_state)

    initial_bias = initialise_bias(output_size)

    {{layer_type, initial_weights, initial_bias}, seed_state}
  end

  defp initialise_weights(col, row, init_type, seed_state) do
    {_, _, _, seed_state, list_of_lists} =
      Enum.reduce(
        0..(col - 1),
        {row, col, init_type, seed_state, []},
        &random_row/2
      )
    {Matrex.new(list_of_lists), seed_state}
  end

  defp random_row(_x, {row, col, init_type, seed_state, acc}) do
    {_, _, new_state, row_vals} =
      Enum.reduce(
        0..(row - 1),
        {col, init_type, seed_state, []},
        &random_val/2
      )
    {row, col, init_type, new_state, [row_vals | acc]}
  end

  defp random_val(_x, {n, :he, seed_state, acc}) do
    {val, new_state} = :rand.normal_s(0, 2 / n, seed_state)
    {n, :he, new_state, [val | acc]}
  end

  defp random_val(_x, {col, :pos, seed_state, acc}) do
    {val, new_state} = :rand.normal_s(0.5, 0.25, seed_state)
    {col, :pos, new_state, [val | acc]}
  end

  defp random_val(_x, {n, :xavier, seed_state, acc}) do
    {val, new_state} = :rand.normal_s(0, 1 / n, seed_state)
    {n, :xavier, new_state, [val | acc]}
  end

  defp initialise_bias(col) do
    Matrex.zeros(col, 1)
  end

  defp layer_to_init(:hidden_layer), do: :pos
  defp layer_to_init(:output_layer), do: :xavier
end
```

Listing 2: Cost Function

```
  def loss(activation, target_activation) do
    target_position = get_target_category(target_activation)

    activation
    |> Matrex.at(target_position, 1)
    |> (fn x ->
         if x == 0.0, do: 100, else: -:math.log(x) end
       end).()
  end

  defp get_target_category(target_activation) do
    {row, _col} = Matrex.size(target_activation)

    Enum.map(1..row, fn x -> [x] end)
    |> Matrex.new()
    |> Matrex.multiply(target_activation)
    |> Matrex.sum()
    |> round()
  end
```

B two