Exploring Distributed Machine Learning System on Raspberry Pi Computer Cluster

Isaac L. Torres Torres
Master's in Computer Science
Alfredo Cruz, PhD
Electrical and Computer Engineering and Computer Science Department
Polytechnic University of Puerto Rico

Abstract *– This project explored the use of Distributed Machine Learning (DML) as a potential tool in training times of Machine Learning (ML) models in lower-end computer cluster, to provide alternatives for students and scientists when implementing their ML environment without expensive/performant hardware. As part of this, an ML training environment was developed and deployed using container technology on a 4-node raspberry pi computer cluster. This cluster was used to train ML classifier models over the popular CIFAR10 dataset. Test cases were set up to analyze how the training times for models were affected when adding and removing nodes from the system and varying the number of processor cores allotted to the system. Data was recorded for each test, such as the test's execution time, average CPU time spent when building the model, overhead, and model accuracy, among others. When analyzing this data, it was found that they were practical limits to the speedup on training times achievable when using DML for the cluster, with diminishing returns on speedup values when adding additional nodes. Meanwhile, the speedup observed when increasing processing power for the cluster displayed no such limitations, showing that DML can be used to improve training times for lower-end devices but in a limited capacity.*

Key Terms: *Containers, Distributed Machine Learning, Docker, Limitations Raspberry Pi.*

INTRODUCTION

Machine Learning (ML) is a section of computer science that involves applying a set of statistics over a group of data to generate a helpful process or algorithm to achieve some goal(s). Some examples of ML applications include controlling self-driving cars [1], recognizing speech [2], predicting market trends [3], among others. Usually, when working with any nontrivial machine learning application, a significant amount of data is required. Machine learning models are valued depending on how accurately they can complete the task, and it is generally the case that models trained with higher amounts of data tend to be more accurate. Although many factors are also involved in this, a significant amount of data is needed to be processed to pursue better and more accurate models. This results in a rise of the necessary processing power required to train models in a reasonable amount of time.

There are two possible ways to approach this scaling problem. The first is to perform vertical scaling. The classic example of this is adding programmable GPUs to a host system. These GPUs feature a high number of hardware threads which improve performance; this has been a proven and tested method [4, 5]. The second way this can be approached is by scaling horizontally. This is where distributed machine learning systems come in; these are systems and algorithms designed to take advantage of multiple computer nodes to process workloads faster than traditional machine learning strategies.

This project had the purpose of viewing how distributed machine learning can be leveraged to allow lower-end devices to be used to complete nontrivial machine learning tasks. This was explored by developing a training environment/system to be used for training machine learning models. This system was used to perform various tests on a microcomputer cluster consisting of 4 Raspberry Pi computer nodes. These tests consisted of training machine learning models as classifiers on the popular CIFAR10 image dataset. This dataset consists of 60000 32x32 color images of one of ten possible classes. The system allowed different configurations to train models on the CIFAR10 test data with a varying number of nodes in the cluster and the number of processor cores used on each host processor. These values were varied to view the impact on performance. The unique combination of these served as the different tests conducted in the project.

LITERATURE REVIEW

Machine Learning

Machine Learning is a subset of computer science derived from the study of artificial intelligence. It can be described as the study and application of algorithms that can learn. There are three main broad approaches regarding machine learning which are: Unsupervised Learning, Reinforcement Learning, and Supervised Learning. The last of these was used in this project. Supervised learning algorithms are those that operate with a set of data that already has all data points classified. In simple terms, humans tell the algorithm what values to look for and which decisions are right. Additional data on which the model was not trained upon is then used to determine if the model makes accurate predictions based on its experience with previous data [6].

Computer Clusters

This is a group of computers that are working or coordinating together to achieve some goal. This is done mainly for two reasons: Improving performance or throughput, or improving uptime by providing a backup computer in case the primary computer fails. This project's main purpose for using a computer cluster is the first reason. Training ML models takes time with more complex models requiring an extensive amount of training before being able to approach an acceptable level of accuracy in their predictions. As such, it is only logical to search for ways to reduce this training time. Increasing the speed of training a model can be done by training on performant hardware. But there are limits to this, a researcher is constricted to whatever hardware is available to acquire and not every computer scientist or company has access to the higher-end components necessary. Meanwhile, computer clusters allow for increasing the count of individual computers in which a program will run to improve performance. This comes at the price of some overhead involved in orchestrating each computer to work together.

Distributed Machine Learning

Next, after discussing Machine Learning and Computer Clusters, the next logical step is how to implement machine learning in a Computer Cluster or distributed environment. Distributed Machine Learning is a term associated with specific M.L. algorithms designed to run multi-node systems, or specific systems designed to improve performance, accuracy or be able to handle large amounts of input data [7].

Figure 1 shows a visual representation of the difference between traditional machine learning and distributed machine learning. There are two main paradigms used when discussing distributed machine learning: data parallelization and model parallelization. In the Data-Parallel approach, the data is partitioned as many times as there are worker nodes in the system. In the Model-Parallel approach, exact copies of the entire data sets are processed by the worker nodes which operate on different parts of the model. For this project, a model-paralleled approach was used.

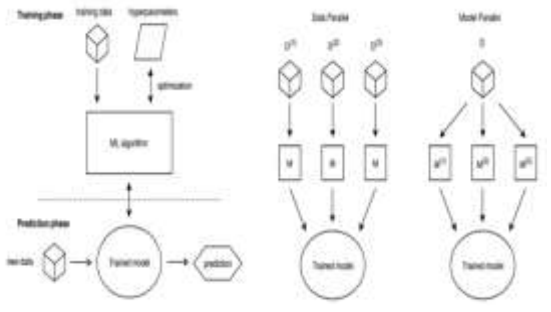

Figure 1
Traditional ML vs DML [8]

This project specifically focused on a computer vision problem which is a subset of ML. Computer vision is defined as a field inside artificial intelligence. Computer vision allows computers and applications to deduce useful information from image data, videos, and other visual inputs and take actions or make recommendations based on that information. This project involved building a simple classifier model for digital images. As such, the type of ML algorithm chosen was a Convolutional Neural Network these are a type of Neural Network that has useful applications for computer vision problems.

Neural Networks

Artificial neural networks are often just called Neural Networks (NN) is a field that investigates how simple models of biological brains can be used to solve difficult computational tasks as seen in machine learning. Neural Networks can be used to make predictions mostly because they can be structured into different forms, and these can have a hierarchical or multi-layered structure to them.

When trying to apply N.N. to solve computer vision problems there are several drawbacks because of the unique qualities of image processing. If designing a N.N. to accept an image as an input, this would require at least one perceptron for each input, the inputs being the number of pixels in the image [9]. For very small images this might be possible, but as an image grows the number of neurons, it would rapidly become unwieldy for large and higher resolution images. It is apparent that for computer vision neural networks are not the best choice for the job.

Convolutional Neural Network

This is where Convolutional Neural Network (CNN) comes into place. A CNN is a Deep Learning algorithm that can take in an input image and be able to determine and assign importance to various aspects/objects in the image. And additionally, be able to differentiate objects one from the other noticing features such as color and shapes while not suffering from the same drawback from using traditional NN [10]. This is done using filters, these are data arrays that are used to parse an image. To apply a filter, of a size specified by the user (typically 3x3 or 5x5 pixels) it is moved across an input image. Typically, from the top left to bottom right. During this process for each point on the image, a value is calculated, this value is computed based on the filter being used to apply a convolution operation. The convolution operation simply consists of passing the filter (also called a kernel) over an input, generally an image. On this pass, when the values of the image match the size of the kernel, matrix multiplication is performed between the kernel and the input, to provide the value of one cell on the output.

Figure 2 shows a 5x5 input matrix (I) which is being applied a 3x3 filter/kernel matrix (K) the first shaded cell in the output 3x3 matrix (O) is the result of perfuming a matrix multiplication of the blue shaded area of the matrix I and the filter matrix K in yellow.

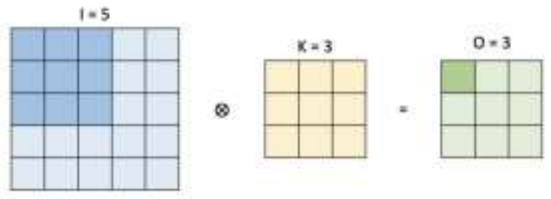

Figure 2
Matrix Convolution Operation [11]

What happens when passing a filter over an image? After a filter has passed over the image in a CNN, a feature map is generated for that filter. These represent patterns recognized in the image. Multiple filters can be applied to an input to produce different feature maps. Still, more importantly, filters can be staked to produce layers of more detailed feature maps, which become more and more abstract as a deeper CNN is created that they can detect more complex features such entire faces, for example.

Training Dataset

The dataset used in this project for training and testing models is the CIFAR-10. This dataset is composed of 60000 32x32 images which belong to one in ten classes, each class corresponds to 6000 respective images. The data set is divided into a training and test set where there are 50000 training images and 10000 test images. The ten classes which appear in the dataset are the following:

Classes: 'plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'

Figure 3 presents a sample of the different images contained in the dataset.

Figure 3
Example of CIFAR10 dataset images [12]

METHODOLOGY

Overview

A system/environment was required which would allow for quickly training and testing ML models. Not only in a single host machine but to also be distributed through multiple hosts, specifically in a computer cluster consisting of 4 raspberry pi computers. The developed solution consisted of a combination of tools and code. As discussed previously the purpose of this was to examine how distributed machine learning could be leveraged in lower-end computer systems specifically a Raspberry Pi computer cluster. This required the following steps:

1. Cluster Set up – The physical setup of a Raspberry Pi computer cluster and all the necessary configurations needed to prepare nodes in the cluster.
2. Code Implementation – This encompassed the implementation of machine learning algorithm(s) capable to be used in a distributed and undistributed environment and the configuration of tools and dependencies involved.
3. Test Case Development & Execution– This step involved establishing and executing a set of different tests to be conducted to compare the distributed and undistributed designs to observe and record results.
4. Analysis of Results – The final from involved analyzing the gathered data from the previous step to draw conclusions and comparisons.

Design

The main code for this project consisted of three main scripts or programs (written in python):

1. Standalone Client – This program was responsible for running the machine learning implementation designed to run on a single host.
2. Server Client – This program was part of the distributed machine learning program used which works in conjunction with the worker-client program or programs to train a model.
3. Worker Client – Second half of distributed machine learning implementation.

Figure 4 shows a representation of how both the distributed and undistributed algorithms were designed to run on the RPI Cluster.

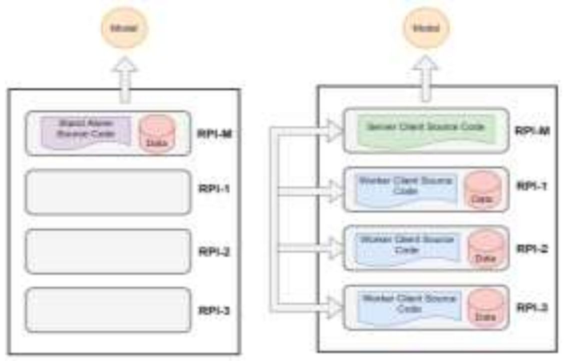
Model
Stand Alone Source Code
Data
RPI-M
RPI-1
RPI-2
RPI-3
Model
Server Client Source Code
RPI-M
Worker Client Source Code
Data
RPI-1
Worker Client Source Code
Data
RPI-2
Worker Client Source Code
Data
RPI-3

Figure 4
Visualization of main programs used in the project

The standalone client consisted of a simple machine learning example that would execute various tasks such as first loading the CIFAR10 dataset into the file system and then dividing the dataset into a testing and training dataset. Then the program would train an ML model over the train set. An additional script was used for testing the resulting model over the test dataset and outputting the accuracy of the model. This program was designed to be run on a single host in traditional ML fashion. When referring to this program further on it will be either as the standalone implementation or the undistributed implementation of the classifier algorithm.

The second and third programs were designed to be executed together in a multi-host environment i.e., the RPI computer cluster. The second was a program that acted as server-client which was designed to run on a master node in the cluster, this program was responsible for assigning work to worker nodes/clients as well as grouping results of work from the worker clients once received and finally creating a single ML model as an output.

The server and worker clients were designed to communicate and operate between themselves using the flower framework, a python library which facilitates implementing distributed machine learning tasks. This library would handle the creation of the output model by aggregating the weights of the different parameters produced from each worker and taking the average between them. Internally the flower framework uses RPC channels to increase the speed of communication between the master and clients' nodes.

Training algorithm: CNN

In this section the code for the machine learning algorithm used and its properties are reviewed. A code snippet with the declaration of the convolution neural network used for training is:.

```
# Class used to define Neural network
class NueralNetwork(nn.Module):
    def __init__(self):
 # Constructor
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
 # Adding 2d Convolution Layer
        self.pool = nn.MaxPool2d(2, 2)
 # A Pooling Layer reduces the variance
        self.conv2 = nn.Conv2d(6, 16, 5)
 # Additional 2d Convolution Layer after pooling
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
 # Linear transformation
        self.fc2 = nn.Linear(120, 84)
 # Linear transformation
        self.fc3 = nn.Linear(84, 10)
# Linear transformation
```

The CNN used consists of 2 convolution layers and 3 Linear transforms:

1. The first layer applies a filter to extract features from the image. These would detect basic lines, curves and basic outlines that are detected on the image.

2. The second layer was a pooling layer which is used to take the average of a region when a filter is passed through producing a layer with reduced dimensions. This output is passed through an additional convolution layer to extract additional more complex features from the averaged previous layer.

3. Finally, three successive linear transforms were applied to the resulting layer. These final transforms reduce the output of the convolution layer to 10 possible outcomes corresponding to the 10 classes of which the images of the CIFAR10 dataset could be. When running an input through the model the output with a higher weight would be chosen as the model’s guess.

Running Test Cases

The Processing of running a test on the system was simplified thanks to the selection of tools used. First, test cases were defined using docker-compose files. These files defined the appropriate images (bundles of code and dependencies) to be used in the test, where containers (instance of the code in image) would be deployed to the cluster. The number of nodes and the number of CPU cores used. When running the test on a single host a single container was sufficient. But for our distributed test using the RPI cluster, multiple containers were necessary which had to run in different nodes in the cluster. Normally, these would have to be started individually, in this area docker-compose files also helped as they replaced having to rely on the Docker CLI for the deployment of our containers since it is possible to manage multiple containers through a single compose file.

The following code sample shows an example of a docker-compose file:

```
master:
    image: ${TARGET}-master-node
    container_name: ${TARGET}-master
    build:
      context: "../"
      dockerfile: "server.${TARGET}"
    deploy:
      mode: global
      constraint: node.label==RPI-M
networks:
  cluster_network
```

The files could be executed in a group of host computers through a docker swarm which in short is a group of computers logically through their docker daemons (running docker process). Each worker node in the cluster needed to be registered to the master node via the docker CLI to achieve this. When running a compose file on the main node of a docker swarm, the docker daemon handles reads the compose files, and starts managing distribution and execution of containers on each swarm node as specified in the compose file used.

As mentioned before, these compose files were used to define and execute the different test cases in the project. When a test case was to be executed by its corresponding compose, the file was fed to the docker client on the master node of the RPI cluster, also known as the manager for the docker swarm. After which, the docker would take care of deploying containers matching the specification given in the compose file. Each container run by docker in this way is considered a service and the group of deployed services are collectively referred to as a stack by docker.

Figure 5 shows a visual representation of the process of building and uploading the final images containing all the necessary source code to execute each test case and how these images were used by each node. For caching purposes, these images were accessed through a remote registry where the images were stored. The benefit of using a registry to distribute images is that the images do not have to be updated locally on each node of the cluster before use.

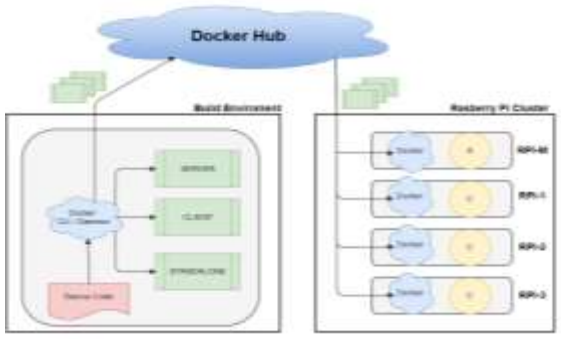

Figure 5
Process visualization of creating dispensing docker images

The process of deploying test cases to the cluster could be managed from a host external to the cluster thanks to docker contexts which allow remote commands to be sent to an external docker daemon. When targeting the RPI cluster the docker context was set to the master/manager node of the docker swarm in the RPI cluster. This way the cluster test could be performed remotely without having to directly access the cluster. Figure 6 shows how test cases were executed from an external host through compose files.

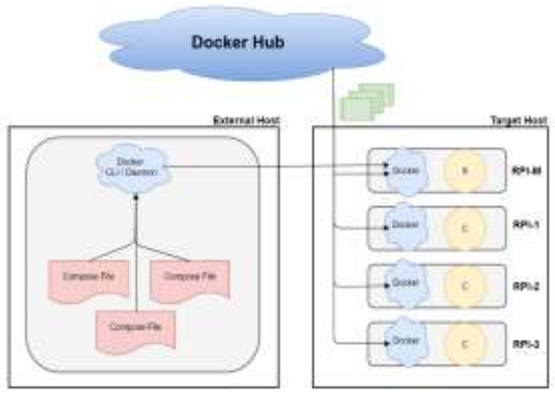

Figure 6
Visualization of Running Test Cases

Data Collection

As discussed previously the testing process was done using composed files. Each compose file corresponded through a different test case which is run through the system. The different test cases consisted of running the same machine learning task in this case Training a classifier on the CIFAR10 data with varying amounts of worker nodes and varying amounts of processor cores. Each possible configuration of nodes and cores had its own unique compose file. For these test cases, various values were recorded and calculated based on the recorded data. To reduce the variance of these values each test case was run three times to produce an average value for test case. For each of these the following values were recorded and/or calculated:

- Total Training time (T_{time}) – This is the time required for training a model using the parameters established in the test case. This value was averaged over three runs of the same test case.
$$\Delta T_{time} = \frac{T_1 + T_2 + T_3}{3}$$
- Total Work Time (W_{time}) – This approximated the time taken to training for the test case. This was calculated as the average work time between all the nodes used in the test case.
$$\Delta W_{time} = \frac{W_1 + \cdots + W_n}{n}$$
- Accuracy (Acc.) –The accuracy for the final model produced by the test case.
$$\Delta Acc = \frac{A_1 + \cdots + A_n}{3}$$
- Total overhead Time (O_{time}) – This was the amount of time spent by the system on communication between nodes. This was calculated by subtracting the total work time from the total training time.
$$O_{time} = \Delta T_{time} - \Delta W_{time}$$
- Work Percentage ($W_{\%}$) – This value represents the percentage of time the system spent performing actual work training the model.
$$W_{\%} = \frac{\Delta W_{time}}{\Delta T_{time}}$$
- Overhead Percentage ($O_{\%}$) – This value represented the percentage of time the system spent communicating between nodes.
$$O_{\%} = \frac{\Delta O_{time}}{\Delta T_{time}}$$
- Speedup – This was calculated through a simple division of training times obtained from test cases. Each test case was compared to a base training time which would be obtained from test case with the lowest number of nodes or cores.

$$\text{Speedup } (S) = \frac{\Delta \mathrm{T_{time}} \text{ (Test Case x)}}{\Delta \mathrm{T_{time}} \text{ (Test Case(base))}}$$

Results

In this section the results of all test cases are compiled as well as various data visualizations based on the recovered data, as well the results of any additional calculations performed. The abbreviations used for variables in the previous section are used reused in this section. The addition of “N” and “C” are short for “node count” and “core count” and represent the number of nodes and processor cores allotted for use when running the test case. Table 1 presents the result of training a model by first varying the number of worker nodes in the system.

Table 1
Results for increasing node counts

N:C	ΔT_{time}	ΔW_{time}	$W\%$	$O\%$	S	ΔAcc
1:2	936.3	936.3	1.00	0.00	0.00	0.56
2:4	478.8	437.6	0.91	0.09	1.96	0.55
3:6	387.0	327.0	0.84	0.18	2.42	0.56

From this table, it can be observed that training times were reduced by adding additional worker nodes. Though, some additional time was lost due to overhead in the system. This however still resulted in positive speed up values after adding additional nodes. Additional nodes could be physically added to determine an inflection point where the overhead involved with communication starts to be greater than the actual CPU time. Due to lack of materials, an inflection point was determined through extrapolating from gathered data this is shown further in figure 9.

Figure 7 shows a visualization of the overall training time and the actual CPU time spent by the cluster on average to train models with varying amounts of nodes.

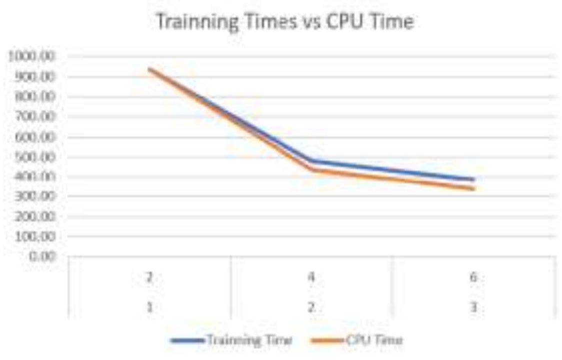

Figure 7
Training times for test cases varying node count

Figure 8 presents the percentage of time each test case spent training a model vs coordinating nodes is presented in the following figure. From the figure, it can be seen the amount of CPU time is slowly decreasing meanwhile the communication time is increasing as more nodes are added.

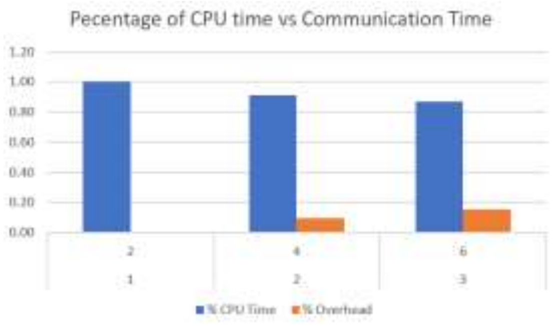

Figure 8
CPU Time vs Communication Time

Figure 9 presents a visualization of CPU training times vs communication time when increasing node count in the cluster.

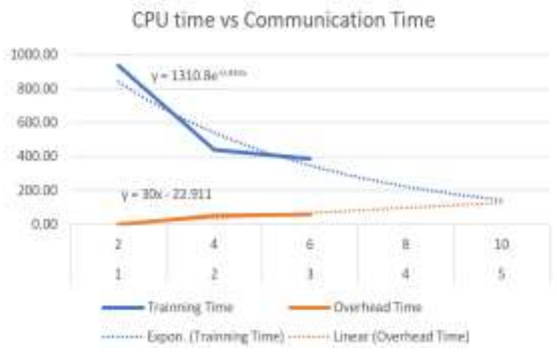

Figure 9
CPU time vs Communication time extrapolated

The amount of processing power used in each node configuration was also varied to measure the impact additional computational power would have on each node configuration. The next table presents test cases that involved a fixed node count but varying the number of processors available to use during training. Specifically, Table 2 shows the results for varying the number of cores used when running the cluster in a 3-node configuration.

Table 2
Results for varying Core count in 3-Node Configuration

N:C	ΔT_{time}	ΔW_{time}	$W\%$	$O\%$	S	ΔAcc
3:1	2205.5	1856.9	0.84	0.16	1.00	0.57
3:2	1247.9	1086.4	0.87	0.13	1.77	0.56
3:3	815.0	726.8	0.89	0.11	2.71	0.59
3:4	628.8	561.8	0.89	0.11	3.51	0.58
3:5	503.2	453.0	0.90	0.10	4.38	0.6
3:6	387.0	327.0	0.84	0.18	5.70	0.61

It can be seen that as more cores were allotted for the cluster to use, training times for models decreased. Figure 10 shows a visualization of training times for test results using a 3-node configuration. Since this configuration contained the most data points linear and exponential regression was used to approximate the behavior of the cluster.

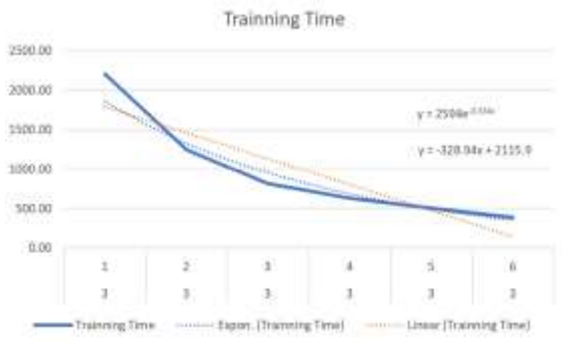

Figure 10
Results for varying Core count in 3-Node Configuration

Figure 11 presents a comparison between the trends for speedup values obtained from increasing CPU count and increasing node count. For this figure speedup, the data from tables 1 and 2 were used to present the behavior of the system when adding computational power to the system without adding more nodes. It is important to note that adding a node is equivalent to adding two cores to the cluster as each RPI has two cores. This figure was created to view if modifying the existing hardware would produce better results than adding additional nodes to the cluster.

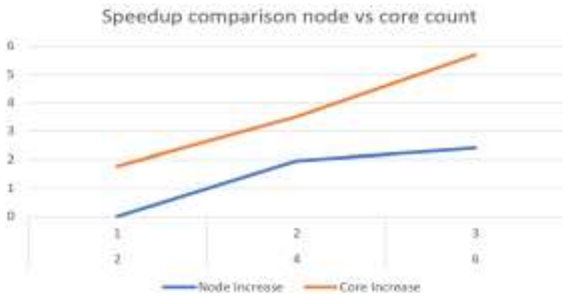

Figure 11
Speedup comparison for varying nodes and cores

Finally, Figure 12 shows a visualization of the accuracy of values for all test cases tried. The average accuracy was calculated to be 57±1.7%.

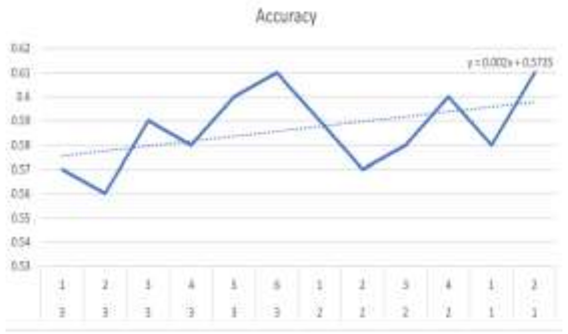

Figure 12
Accuracy of Models across all test cases

DISCUSSION

From the results, various trends regarding the recollected data can be seen. First is accuracy for all models produced by test cases which remained almost constant hovering around 57±1.7%. The is strong evidence that training models in distributed environments have little effect if any on the accuracy of trained models. Second training times for models when adding additional nodes outpaced training times for models trained by adding additional cores. At least meanwhile, the number of cores remained low. Also, from observing the behavior of speed up values it was found that adding additional nodes has diminishing returns when compared to simply adding more processing power to the system. From figure 9 it can be inferred that this limit lies around 4 to 5 nodes when the overhead time starts to become greater than the actual time spent performing work. This is consistent with current literature regarding how coordination and communication time between nodes is one of the more significant bottlenecks in distributed environments.

CONCLUSION

For this project, an environment for training machine learning models with the functionality to scale up or down with relative ease was successfully created. This system was used to examine the performance of an RPI cluster in training classifier models over the CIFAR10 dataset. By observing training times produced by the test cases, it was found that in general, models produced using a distributed approach were trained in less time than models trained with an undistributed approach, at least for initial test cases, which involved low processing power. Additionally, when examining the effect of adding additional cores to the system without the added complexity of adding additional nodes it was found that this could result in greater speedup values when adding the equivalent processing power of an additional node to the system. It is apparent that training machine learning models was feasible in an RPI. Although, examining recovered data it was observed that training time could not be reduced indefinitely by adding additional nodes to the cluster, due to diminishing returns. For this particular implementation, the max practical number of RPI that could be used in the cluster was found to be from 4 to 5. Since RPI are not traditionally designed to have their hardware be upgraded, this creates a hard limit for workloads able to be run on RPI clusters.

Acknowledgments

This work is supported by, or in part by, the DoD Cybersecurity Scholarship Program (CySP) under grant # H98230-20-1-0355. Special thanks are extended to Dr. Alfredo Cruz, Ph.D. of the Polytechnic University of Puerto for his extended supervision, guidance, and patience throughout this project.

References

[1] M. Bojarski, et al. (2016). *End to End Learning for Self-Driving Cars.* [online]. Available: http://arxiv.org/abs/1604.07316

[2] D. Amodei, et al. (2016). *Deep speech 2: End-to-end Speech Recognition in English and Mandarin*. PMLR. [online]. Available: http://proceedings.mlr.press/v48/amodei16.html

[3] A. Khandani, A. Kim, and A. Lo,. “Consumer Credit-risk Models via machine-learning algorithms,” *Journal of Banking & Finance*, vol. 34, issue 11, pp. 2767-2787, 2010.

[4] H. Kargupta, et al, “MobiMine: Monitoring the Stock Market From a PDA,” *ACM Explore. News.*, vol. 3, no. 2, pp. 37–46, 2002.

[5] H. Kargupta, et al, “VEDAS: A Mobile and Distributed Data Stream Mining System for Real-time Vehicle Monitoring,” In Proceedings of the 2004 SIAM International Conference on Data Mining, Lake Buena Vista, FL, USA, pp. 300–311, 2004.

[6] J. Hu, H. Niu, J. Carrasco, B. Lennox, and F. Arvin, “Voronoi-Based Multi-Robot Autonomous Exploration in Unknown Environments via Deep Reinforcement Learning,” *IEEE Transactions on Vehicular Technology*, vol. 69, no. 12, pp. 14413–14423, 2020.. doi:10.1109/TVT.2020.3034800

[7] P. Tean. (2020). *Distributed Machine Learning for Big Data*. [online] Available: https://www.guavus.com/technical-blog/distributed-machine-learning-for-big-data-and-streaming/

[8] L. Mao. (2019). *Parallelism VS Model Parallelism in Distributed Deep Learning Training.* [online] Available: https://leimao.github.io/blog/Data-Parallelism-vs-Model-Paralelism/

[9] J. Verbraeken, et al. (2020). *A Survey on Distributed Machine Learning*. [online] Available: https://dl.acm.org/doi/fullHtml/10.1145/3377454

[10] P. Walpita. (2020), *Convolutional Neural Networks for Artificial Vision.* [online] Available: https://priyalwalpita.medium.com/convolutional-neural-networks-for-artificial-vision-455be7c85d15

[11] S. Saha. (2018). *A Comprehensive Guide to Convolutional Neural Networks*. [online] Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[12] A. Krizhevsky. (2013). *CIFAR-10 and CIFAR-100 Datasets*. [online] Available: https://www.cs.toronto.edu/~kriz/cifar.html