Exploring Distributed Machine Learning System on Raspberry Pi Computer Cluster

Isaac L. Torres Torres, Masters Student
Polytechnic University of Puerto Rico
isaaclive@gmail.com

Alfredo Cruz. PhD
Polytechnic University of Puerto Rico
alcruz@pupr.edu

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 clusters. 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 (RPI) computer cluster. This cluster was used to train ML classifier models over the popular CIFAR10 dataset. Several 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 processing power, i.e., 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. This showed that although DML can be used to lower training times. This showed that DML can be used to improve training times for lower-end devices but in a limited capacity.*

Keywords: Distributed Machine Learning, Raspberry Pi, Limitations, Containers, Docker.

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 (Bojarski, M. et al. 2016), recognizing speech (Amodei, D. et al. 2016), predicting market trends (Khandani, A., Kim, A. & Lo, A. 2010), among others. Usually, when working with any non-trivial 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 (Kargupta, H. et al. 2002, Kargupta, H. et al. 2004). Similar methods mostly revolve around a similar concept of adding additional specialized hardware to a single host system. 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. The benefits of such a strategy have been observed and replicated by research such as "*A holistic approach for resource-aware adaptive data stream mining*" (Philip, S.Y. 2006).

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 (RPI) computer nodes. These tests consisted of training machine learning models as classifiers on the CIFAR10 dataset. This dataset consists of 60000 32x32 color images of one of ten possible classes. The main quantifiable properties of the models which were focused on were training times and accuracy. 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.

Methodology

For this project, a system or 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 environment consists of a combination of tools and code. The system built for this project emphasized the following qualities: simple to install, operate, and horizontally scalable. The system was intended to have "Plug and Play" functionality and be able to be run by users as soon as installed. By horizontally scalable it is referred to how the computational capacity of the environment or system can be increased or decreased with relative ease by adding or removing nodes.

As part of the development of this system the following steps were required:

1. Cluster Set up – This is 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 encompasses the de-implementation of a machine learning algorithm(s) capable to be used in a distributed and undistributed environment and the configuration of used tools.
3. Test Case Development & Execution– This step involved establishing a set of different tests to be conducted to compare the distributed and undistributed implementations and viewing and recording observed results.

4. Analysis of Results – The final from involved analyzing the gathered data from the previous step to draw conclusions and comparisons.

The main code for this project consisted of three main scripts or programs (using the python programming language) these were:

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.

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. Worker nodes were nodes running the final program that would be run in any node of the cluster other than the master node. This was reserved always just for the server-client program.

Figure 1 shows a representation of how both the distributed and undistributed algorithms were designed to run on the RPI Cluster:

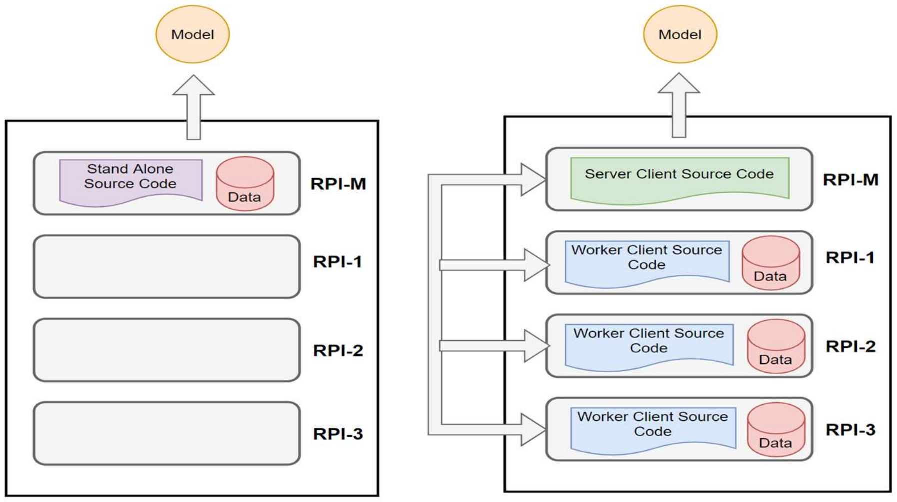
Model
Stand Alone Source Code
Data
RPI-M
RPI-1
RPI-2
RPI-3
Model
Server Client Source Code
RPI-M
Worker Client Source Code
Data
RPI-1
Worker Client Source Code
Data
RPI-2
Worker Client Source Code
Data
RPI-3

Figure 1: Visualization of main programs used in the project.

Overall, there was always at least one instance of the server-client running on in the master node and one to three server clients running on each worker node in the cluster which consisted of 4 RPI model 4 B computers. 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.

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 for deploying an instance of the server-client to the master node of the cluster:

```
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 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 docker would take care of deploying containers matching the specification given in the compose file. This included specifying the number of nodes and processor cores allotted for use in the test. Figure 2 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.

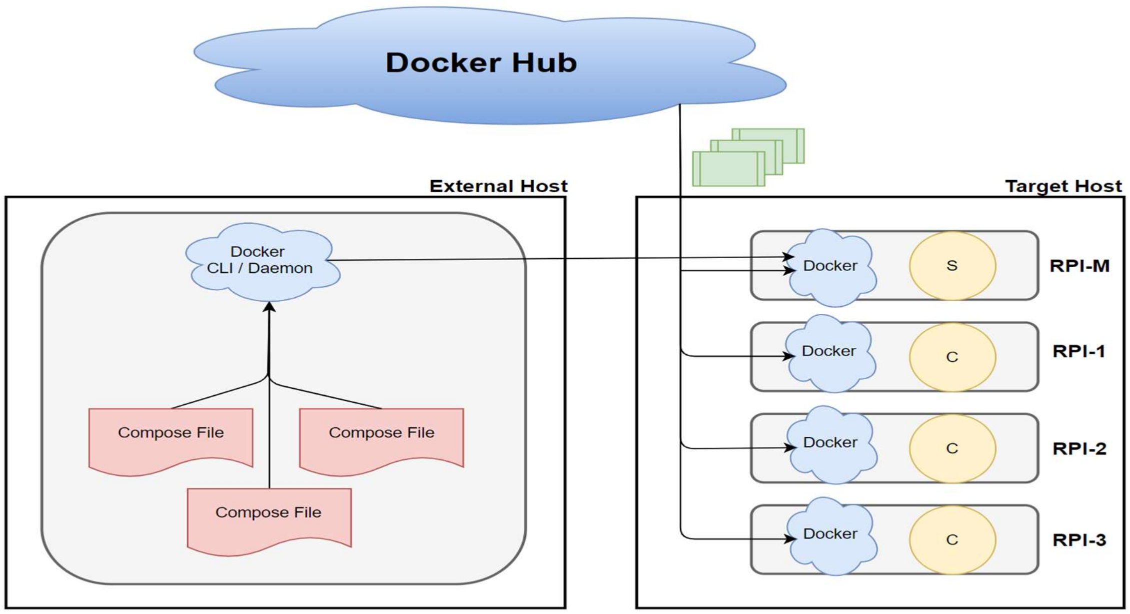

Figure 2: Visualization of Running Test Cases.

Next the machine learning algorithm used, and its properties are reviewed. Below is a code snippet with the declaration of the convolution neural network used for training.

```
# 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 tranformation
        self.fc2 = nn.Linear(120, 84)
 # Linear tranformation
        self.fc3 = nn.Linear(84, 10)
# Linear trnsformation
```

The CNN used consists of 2 convolution layers and 3 Linear transforms. 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. The second layer is 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. Finally, three successive linear transforms are 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.

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 aggregate of values across three runs. For each test case 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.
- Total Work Time (W_{time}) – This approximates the time taken to complete all training operations for the test case. This is calculated as the average work time between all the nodes used in the test case.
- Total overhead Time (O_{time}) – This is 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.
- Work Percentage ($W_{\%}$) – This value represents the percentage of time the system spent performing actual work training the model.
- Overhead Percentage ($O_{\%}$) – This value represents the percentage of time the system spent communicating between nodes.
- Speedup - To check how each test case compares to the following case the speedup was calculated through a simple division of training times.

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. Next, 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}	$W\%$	$O\%$	$\boldsymbol{S}$	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 seen 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 4. Next Figure 3 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.

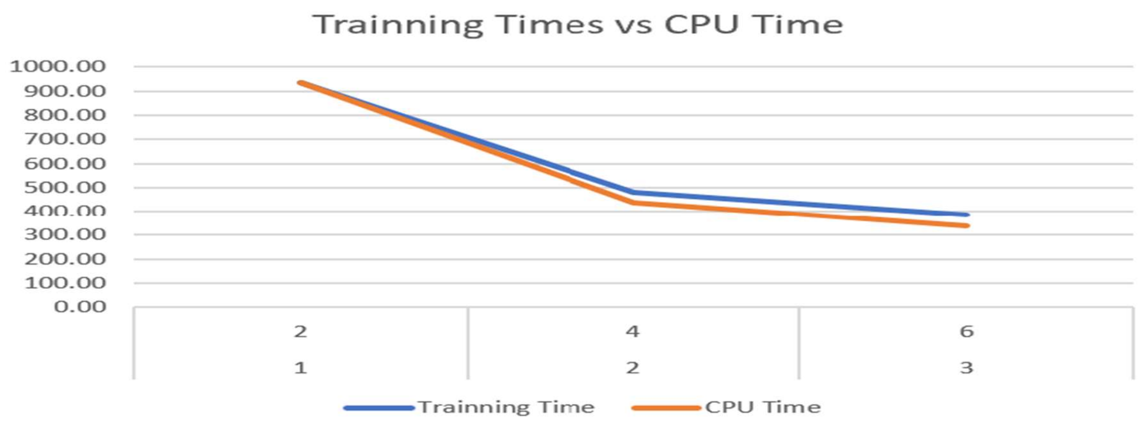

Figure 3: Training times for test cases varying node count.

Next, figure 4 presents a visualization of CPU training times vs communication time when increasing node count in the cluster.

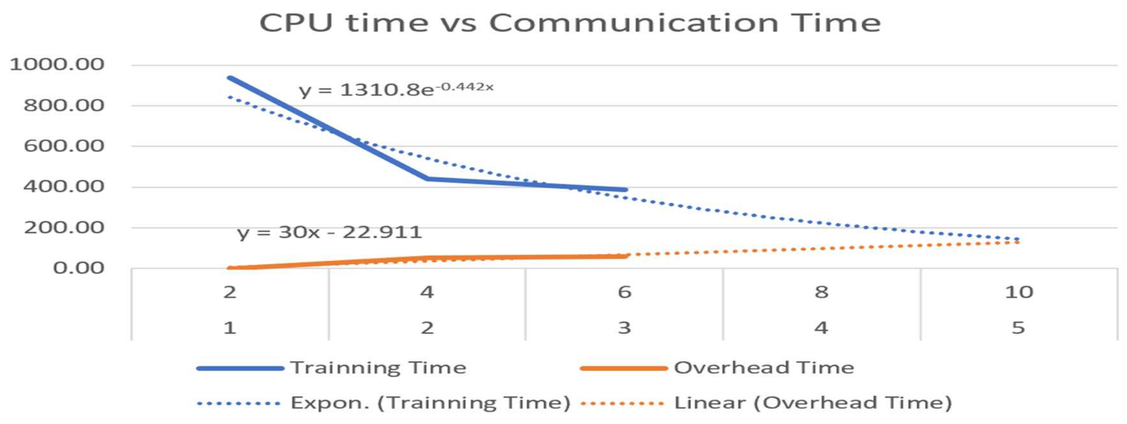

Figure 4: CPU time vs Communication time extrapolated.

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 following tables present test cases which involved a fixed node count but varying the number of processors available to use during training. Next, 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}	$W\%$	$O\%$	$\boldsymbol{S}$	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

From this table, it can be seen that as more cores were allotted for the cluster to use, training times for models decreased. Next, figure 5 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.

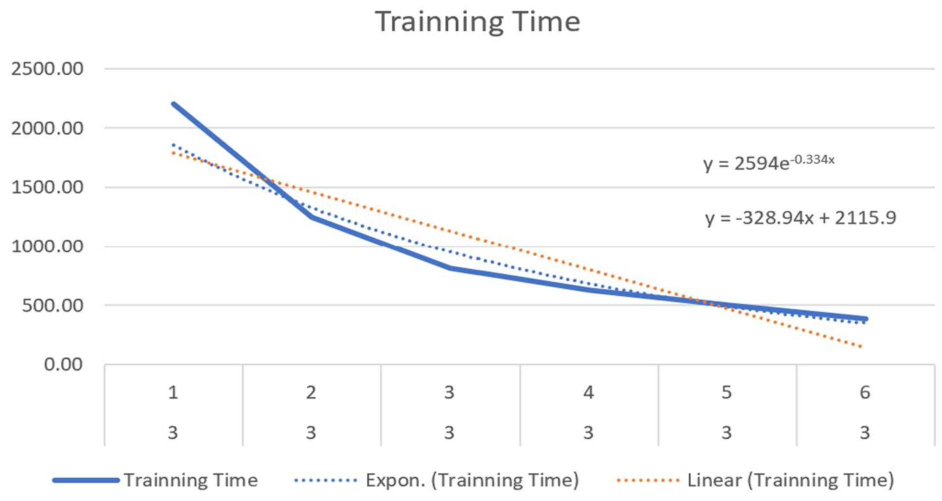

Figure 5: Results for varying Core count in 3-Node Configuration.

Figure 6 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.

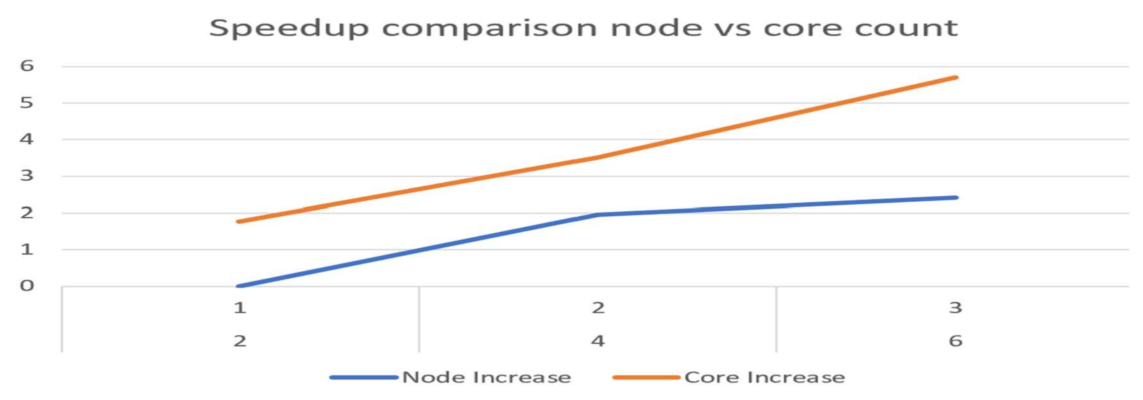

Figure 6: Speedup comparison for varying nodes and cores.

Discussion

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. This is a strong indication that there is a hard limit to the potential reduction of training times that can be achieved using this implementation of an RPI cluster simply by adding additional nodes. From figure 5 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, although due to the low number of nodes used this did not reduce training times significantly.

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 machine learning models over the CIFAR10 dataset. By observing training times produced by running the test cases on the RPI cluster, it was found that in general, models that were 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 lower amounts of cores meanwhile 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 an undistributed system. It is apparent that training machine learning models were feasible in an RPI cluster but there were some limitations. For example, examining recovered data, it was found 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

Amodei, D. et al. (2016). *Deep speech 2: End-to-end Speech Recognition in English and Mandarin*. PMLR. Retrieved January 30, 2021, from http://proceedings.mlr.press/v48/amodei16.html

Bojarski, M. et al. (2016). *End to End Learning for Self-Driving Cars.* Retrieved January 25, 2021, from http://arxiv.org/abs/1604.07316

Kargupta, H. et al. (2002). *MobiMine: Monitoring the Stock Market From a PDA.* ACM Explore. News., 3, 37–46.

Kargupta, H. et al. (2004). *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, 22–24. 300–311.

Khandani, A., Kim, A. & Lo, A. (2010). *Consumer Credit-risk Models via machine-learning algorithms*, Journal of Banking & Finance, 34, issue 11, 2767-2787.

Philip, S.Y. (2006). *A Holistic Approach for Resource-aware Adaptive Data Stream Mining.* New Gener. Comput., 25, 95–115.