Axon: Distributed Machine Learning in Edge Networks

Surjith Bhagavath Singh, Shyam sundar Ramamoorthy, Andrew Candelaresi, Austin Longo

Department of Computer Science, University of Colorado Boulder

Surjith.BhagavathSingh@colorado.edu

Shyam.Ramamoorthy@colorado.edu

Andrew.Candelaresi@colorado.edu

Austin.Longo@colorado.edu

Abstract

Resource requirements at cloud compute datacenters will continue to grow rapidly over the next decade. Meanwhile, an increasing number of low-power computing devices are becoming a pervasive standard at the edge of many networks. These devices are often left underutilized and offer a great deal of resources for data processing and machine learning applications. In this paper, we propose a new framework, named Axon, to facilitate the utilization of these untapped resources for distributed machine learning. We outline the framework components and demonstrate some sample use cases. We ultimately conclude that Axon can outperform local, standalone configurations for certain use cases and could provide a cost benefit to datacenter operations on a large scale.

***Keywords*— Distributed Network, Offloading, Machine Learning, Edge computing, Cloud Computing**

I. Introduction

The amount of data sent to the cloud for processing and computing has exploded over the past decade and a half. The increase in IoT devices, smartphones and social media has produced a flood of data. Performing analytics on all of this data requires an ever increasing amount of computational resources. The most compute intensive part is in the earlier stages where data dimension is high. If the initial steps can be offloaded to edge devices, there is the potential to save a lot of computational cloud resources. Clients could volunteer their devices in an approach like a bittorrent network, where a set of nodes can register to do some part of the task. Users would be incentivised to join for a small financial compensation based on the time their device is used. We can find additional resources by harnessing lightweight edge nodes. In our research we examine running machine learning algorithms and doing data processing on these distributed edge environments and propose our framework, Axon, as a plausible solution to improving efficiency in edge computing.

II. Motivation

In recent years there has been significant progress made in running advanced machine learning algorithms in a distributed network [11]. Leveraging massive clusters of machines using cloud architecture datasets that are truly massive can be processed through the use of distributed systems. Parallelising the intensive computational portions of machine learning has provided massive advantages and allowed researchers to study problems with much higher dimensions features and use larger data sets. Frameworks like Spark and Tensorflow have these distributed architecture built in.

There has been much research done concerning data consumption at cloud data centers centers[13], [14]. It is predicted that data centers will triple in their energy consumption over the next decade [15]. One strategy for alleviating some of this excessive energy consumption is to offload some of the data processing onto edge clusters where energy consumption gaps can be exploited.

Our framework has been partially inspired by the BOINC platform, in which people all around the world donate their computer's unused CPU power to scientific research on massive datasets [17]. BOINC Projects like Asteroids@home and Einstein@home are available on the BOINC android application. In this platform we find an example for utilizing resources that otherwise would have been wasted and therefore saving energy consumption at cloud computing resource centers.

In our implementation we will have to tackle the problem of distributed resource management. One framework that does this eloquently is Blockchain, a distributed ledger. Similar to p2p network, anyone can contribute their resources by joining the cluster. Blockchain maintains the peer table in each peer rather than a centralized table in p2p network. Block chain utilizes one way hash functions to confirm data transactions.

At Spark East Summit Christopher Nguyen discussed the trade off between communication overhead and distributing data across a cluster [12]. His presentation shows that as the

size of the dataset increases and the complexity of the model increases, more time is spent communicating the model and this communication time dominates the performance of your distribution. This bottleneck seems to hold until the dataset becomes so large that it cannot be held on a single machine and distribution is required for processing.

III. Platform and Implementation

Platform for project analysis

We have developed a prototype of our framework Axon, using t2.micro ec2 instances for scalability. Our purpose in doing this is to demonstrate proof of concept. We have registered these clusters with our main super node; which services client requests that are geographically distributed across the Boulder metro area. Fig 1 illustrates our prototype framework.

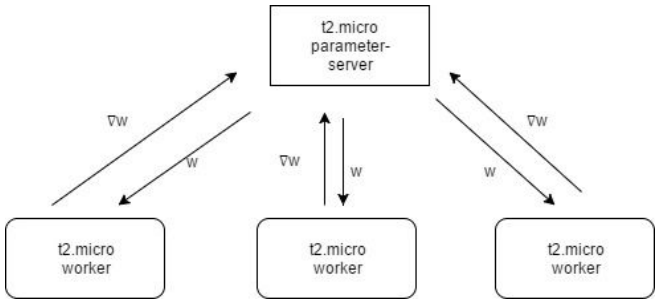

Fig 1. Axon platform architecture

Implementation

In addition to using pre-existing distributed computation frameworks, such as Spark, and Tensorflow, we have also implemented our own basic framework -- Axon. Axon acts as a proof of concept for the effectiveness and correctness of offloading data processing and machine learning to a distributed system. We will from here on refer to the individuals adding their resources to Axon as peers and the individuals using our service as clients.

Peer to Peer

Our framework concept was inspired by bittorrent networks. Instead of making it another distributed implementation of Tensorflow, we came up with the solution of implementing Axon as an autonomous, distributed framework. The user doesn't have to find the IoT devices and input the IP address and port numbers.

Whenever a peer wants to contribute their machine, the peer simply installs the daemon script that we implemented. This script will send http request/posts on a shared table. For prototyping purposes this table is centralized. All the available devices and their computing power are posted in the central shared table.

If the client wants to utilize some of the clusters for training, the client can just send an http request for available devices, for *n* number of peers where their algorithm can be implement and run. The centralized table returns the available devices. An RTT test can be done before acquiring the cluster. This will make sure the acquired device's round trip time is within acceptable limits.

Cluster Configuration

A simple shell script will be deployed which installs the dependencies for any machine learning or data processing tools required by the client and our custom built table formation protocol. This will include the default server addresses to ping to gather facts about the local servers. Future work will include utilizing an automated framework like Ansible, Puppet or Chef that would allow us to configure a cluster remotely based on client specifications.

Shared Table

Resource management tools, like YARN, were looked into. For our application, however, it was easy to just implement a simple shared table that is updated every 20min. This shared table will have information like available CPU resource, Bandwidth, GPU resource, available installed packages with IP and port information.

The table is updated as long as a connection to the system is active. If the connection is closed, then the entry will be removed from table. This gives an up to date information about the nodes that are available. If the system is dead, then it is easy to identify that the system has not updated the table. This table keeps updating its local peers (Synchronizes every 120 minutes). If the client who is using the cluster sees that a node is not responding, it updates the entry in the table as not responding.

Since the resources are labeled with the type of packages that are already installed, this will reduce the custom configuration our system will be required to do.. Our system will be extrapolated to normal PCs with GPUs and open source server clusters.

Future implementation will include an option, whether the sites allow the users to install their custom package, it will be an entry in the table. So the user can custom install the packages they needed.

Peer Server

This shared table is implemented by having a Peer service, running a server always listening on a port. There are redundant servers in case of failure, which can be used to get the information about the sites.

Peer Site

Sites are those that want to offer compute resources with a set of pre installed packages. These clients have a daemon

script running a client, which pings to the Peer servers and post its information. The information includes the port it is listening to in order to serve the user directly. It can serve multiple clients at a time, as well, by listening to multiple ports and depends upon the CPU usage. The client code is listening for the commands from the peers/users directly for the codes to run on the site. Once it gets the code and what command has to be used to execute it, a system command has been initiated and the log has been sent to the user.

User Client

Clients are individuals or groups who need compute resources. Clients ping directly to the Peer Server and get the information about the compute resource. Then this directly contacts the Peer Site and sends the files it needs to execute. Once the files are transferred, the user client talks to the Peer Site and sends the execution command. While communicating the user needs a UUID, which has been generated by the Peer Server, which can also be used to bill, in the future implementation.

Security in Resource Allocation

Whenever an IP address and a port number contacts a Peer server, unique UUID has been generated and it has been stored in the table along with IP information. This UUID and IP address of the User who requested for the resource will be sent from the Peer to Peer Site. When a user contacts a Peer Site, it has to use the IP address it used to ask the Peer Server, and UUID which has been assigned for the User Client. If these two information matches, the Peer Site will send a message that it is authenticated. This could be an important security measure in the system. This is shown in Fig 2.

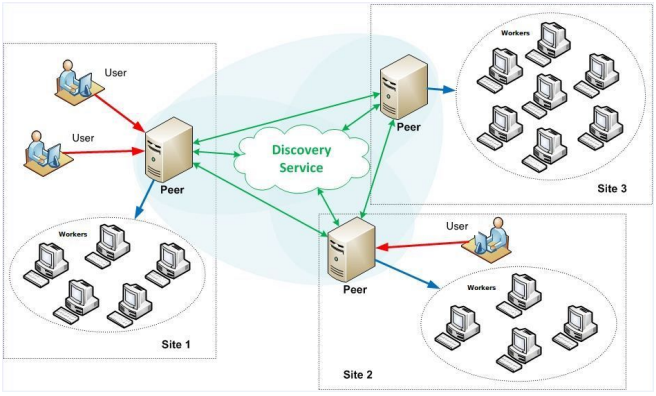

Fig 2. Peer to Peer network for shared table formation

Monetization

The Peer-to-Peer system will work better if it has a monetary incentive. This idea is similar to Bitcoin. If the user is contributing their device, they will receive a small monetary reward. If the consumer (the coordinator) wants to use a big resource, a small amount of money can be involved in making the system accountable. A score system can be maintained for each Peer Site to track the amount of computing resources each Peer Site has contributed and credit them appropriately.

Security

Security is a big concern when we put out a cluster of devices without any of the security features. Especially in a monetized system, security among networked devices is crucial to the success and viability of the service. This aspect of the project is slightly out of scope and is for future development.

IV. Off-loading Machine Learning and Data Processing

We have done our research using Neural Networks, Mapreduce, and other data processing algorithms. We trained a neural network in a distributed environment using Tensorflow. Data was sent to the master node from a user, either cloud or client, and distributed across the worker nodes. A great deal of previous work has been done with neural networks concerning distributed training [1], [2], [6]. Our goal is not to provide a new approach to network training but to examine the performance of these algorithms in our framework.

We chose to use neural networks because the models are computationally intensive to train and were good candidates for Axon. Deciding how many hidden layers and perceptrons a network should have is an important question when designing a neural network, since it increases the computational intensity of training the model. Reference [7] shows that the error in the prediction decreases as the number of layers increases until a point where the number of perceptrons is less than the number of hidden layers. At which point the error increases. This type of indepth network analysis and considerations was decided to be outside of the scope of this project, since the focus was on our framework and not one particular machine learning problem.

Our original plan was to use support vector machines as well as neural networks, but in the process of developing our project it became apparent that support vector machines were not going to be a useful approach. This is because support vector machines are not as easily distributed for training.

Small Datasets

Small dataset like MNIST, Boston house price dataset can handle by the single core or multi core processor system like raspberry pi. Dataset is in the range of 0-100MB of size. If we train a big data model on a single/multi core processor. It will not have enough computing power to train it faster.

Medium Datasets

These are the moderate datasets, size vary from 100MB - 1GB of data. This can be easily handled by GPU acceleration. Here the processing of data has to be higher than data transmission (to be efficient).

Big Datasets

These are the big data sets like facebook and youtube. It is the range of 10GB - 100's of TB of data. This is where distributed machine learning plays a major role. If we train a small dataset in a distributed fashion, one thing we are not utilizing its capacity. Another thing network latency plays a major role here.

V. Distributed Computation Frameworks

Spark MLlib

Apache Spark is an open source distributed computing platform; its main abstraction is Resilient Distributed Datasets (RDD). MLlib is a machine learning library that comes with Spark. It provides tools like algorithms, pipelines and featurization. ML pipelines in MLlib provide APIs that help users to create and tune ML workflows (creating dataset, finding feature vector, learn a prediction model using feature vectors and labels). Each stage in the pipeline is run in order and the input data frame is transformed as it passes through each stage. In addition, Kafka can be combined with Spark to build real-time streaming data pipelines that process (transform) data streams between systems.

YARN

YARN is used to run Spark in the cluster mode. When a job is submitted in Spark with master as YARN, Spark driver will contact YARN resource manager to get Spark executor which is free. Then Spark driver will assign task to executor. Our experiment has 3 executors and 1 driver. YARN thread is spun with help of hadoop. Comparisons are made while running Spark job in client mode (single pi) and cluster mode (multiple pi's) .

Tensorflow

Tensorflow is Google's machine learning library. Its main abstraction is a tensor. A tensor is a node in a graph that defines and stores a series of computationally intensive operations to be executed in a session. To do efficient numerical computing in Python, we typically use libraries like NumPy that do expensive operations such as matrix multiplication outside Python, using highly efficient code implemented in another language. Unfortunately, there can still be a lot of overhead from switching back to Python every operation. This overhead is especially bad if you want to run computations in a distributed manner, where there can be a high cost to transferring data.

Tensorflow also does its heavy lifting outside Python, but it takes things a step further to avoid this overhead. Instead of running a single expensive operation independently from Python, Tensorflow lets us describe a graph of interacting operations that run entirely outside Python [16].

In [9] they discuss the advantages provided by Tensorflow libraries for supporting distributed machine learning. Tasks are split then scheduled on worker nodes in a cluster. Data is returned to the parameter server where it is combined to form a model. This platform is one of our primary candidates, because it has a lot of support from the research community. It has a lot of sample models we have looked into..

VI. Project Evaluation

The evaluation of our distributed data processing and machine learning framework was done via three main metrics; speed, accuracy and latency.

Cluster Formation

Our initial design was to use a cluster of 4 Raspberry Pi devices at different locations in Boulder set up with public IP access and installed with Tensorflow packages. Do to limited financial resources in creating larger Raspberry pi clusters, ultimately, our cluster consisted of all t2.micro instances in Amazon's EC2 environment. This cluster consisted of 10 Tensorflow instances, 5 Spark slaves with 1 master, and 4 Hadoop instances.

Speed

Speed of the system will be largely dependent upon the underlying network connections and the efficiency (or lack thereof) of our message passing protocol (i.e. how much additional network traffic our platform will generate).

Distributed runs will be compared against localized runs (one machine) to determine the best ways to tune the framework for increased processing times and overcome the network latencies.

Accuracy

This metric will be algorithm dependent. Upon execution of each algorithm across our distributed framework, we will evaluate the results against some set of control results to judge how close to accurate they are. Since every machine learning algorithm has slightly different goals, each result will be judged independently in the hopes that a common characteristic of the system can be deduced.

Latency

The speed and accuracy are not the only things we analyze in the model. Network latency plays a major role in deciding whether there is a need for distributed machine learning. If the dataset is small like MNIST dataset, it probably doesn't need

distributed machine learning. When the dataset grows large enough, like the video data that youtube has, the distributed architecture is more performant. A simple comparison from previous work [12], shows that GPU acceleration is better in cases where the data is not extensive enough to distribute but big enough to be accelerated in a GPU.

Limitations

Raspberry Pi has limited memory, so it will handle small and medium datasets. But with large datasets, computation becomes slow. Hence we have limited ourselves to use medium dataset for performance analysis. The numpy installations crash out because of memory error. We are thinking about making big dataset small enough to distribute away to these clusters.

One important concern when working with edge nodes is the very limited memory capacity. Running a machine learning algorithm on a pi without distributing the data. The svm algorithm, as expected, performs well on a single machine when the data set is small to medium. Comparing a non-distributed svm test run on a raspberry pi 3 and a laptop with an intel i7 processor and 6 cores, the laptop performs significantly better the pi took 160 seconds on average vs the laptop took an average of 8 seconds. Based on this we can see that the pi's have very limited computational power and it is only through distribution to several pi's that we gain any computational advantage.

VII. RESULTS

Fig 3a. Time results from running neural network on using Tensorflow on MNIST data. This chart shows that as the size of the cluster grows more time is required to train a model.

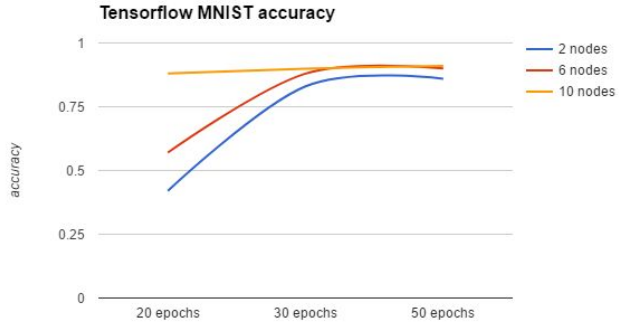

Fig 3b. Accuracy results from running neural network using Tensorflow on MNIST data. This chart shows that as there is a significant increase in accuracy to be gained with larger clusters when the number of epochs is low, but this effect is decreased as the number of epochs increase. Also it should be noted that as the accuracy gets better, it takes many more epochs to gain a single percentage improvement.

We have experimented with several simple problems and datasets. Our first approach was the well known problem of handwritten digit classification using neural networks written in Tensorflow. We were able to use our framework to train a model using two, six, and eight worker nodes. You can see our results in figure 3a and 3b. From the results we can see that there is a trade off between accuracy and training time. As the number of epochs increase the accuracy increases training also increases, but we also see that we attain a significantly improved accuracy with a larger cluster for a lower number of epochs.

Our second approach was performing logistic regression using stochastic gradient descent on increasing input sizes. Datasets consisted of 10 columns (features) of floating point values and ranged from 322 rows to ~780,000 rows (~1MB to 90MB). We were able to use Axon to perform logistic regression across one, two, four and eight nodes. The results can be seen in figure 4a. They show that this dataset analysis experiences speedups when moving from 1 to 2 nodes, but quickly begins to deteriorate with increasing cluster sizes as network traffic overhead becomes prevalent.

This analysis was performed with and without the use of the Axon framework for node discovery and communication. Runs without Axon were simply pre-configured and run manually from the Spark master instance. It was observed that the overhead was at most an increase in runtime of about 1sec, as shown in Fig 4b. When using higher node counts (ie. 8 nodes), the overhead became negligible and there was an indistinguishable difference in run times.

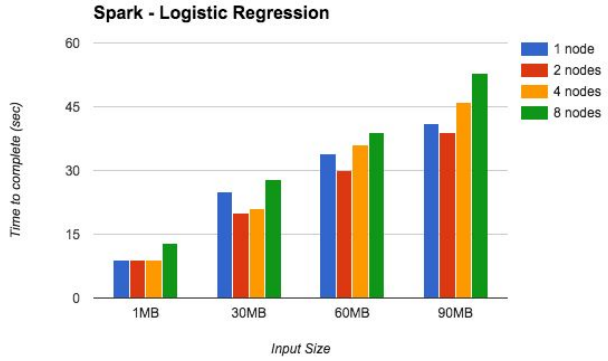

Fig 4a. Time to complete a logistic regression with Spark across varying datasets and with increasing cluster size. Two nodes was the optimal environment for this dataset analysis, striking a balance between distributed compute power and network latencies.

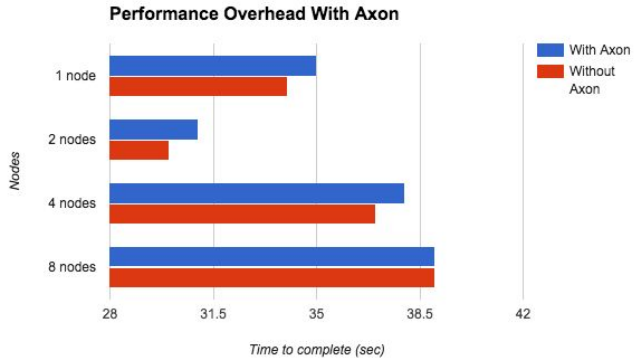

Fig 4b. Performance overhead measured using Axon when running 60MB dataset through logistic regression on an increasing cluster size.

VIII. Conclusion

In our research we have reached a few main conclusions about implementing a practical distributed offloading solution. First distribution of data processing across a large cluster only provides significant performance improvements when a dataset is very large. This statement, however, may also depend on the "shape" of the data and the type of analysis being performed. Second, through monetary incentives we can increase user participation in our framework. Lastly, there is a trade off, when distributing machine learning, between time and accuracy. As the number of nodes increase the accuracy increases, but so does the time it takes to train the model.

IX. Future Work

One interesting area of research is the use of graphical processing units GPUs, built in to most devices, for extra computational power [8]. Raspberry Pi's have an onboard GPU but it is very difficult to gain access to these extra cores. If we can leverage these cores we could parallelise our algorithms significantly. Currently, Tensorflow is only supported on Nvidia GPU's so we will have to wait for more broad support of GPU's. The raspberry pi 3 has a Broadcom VideoCore IV gpu and samsung uses ARM's Mali series gpu. There are reports that Samsung is in talks with Nvidia about potentially getting Nvidia gpu's for there phones in the future. This would greatly increase the ability to parallelize edge node computation on phones.

Additionally, more work should be performed to package Axon to make it more API friendly. This would allow for greater ease-of-use and increase portability of our framework across platforms and devices.

X. References

[1] M. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized stochastic gradient descent. In NIPS, 2010.

[2] Augmenting Operating Systems With the GPU, 1st ed. Salt Lake city: University of Utah, School of Computing, 2012, pp. 1-5.

[4] Y. Radhika and M. Shashi, "Atmospheric Temperature Prediction using Support Vector Machines", International Journal of Computer Theory and Engineering, pp. 55-58, 2009.

[5] C. Szegedy et al., "Going deeper with convolutions," 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, 2015, pp. 1-9. doi: 10.1109/CVPR.2015.7298594\

[6] A. Perez-Vega, C. Travieso, J. Hernandez-Travieso, J. Alonso, M. Dutta and A. Singh, "Forecast of temperature using support vector machines", 2016 International Conference on Computing, Communication and Automation (ICCCA), 2016.

[7] Your Bibliography: [3]K. Abhishek, M. Singh, S. Ghosh and A. Anand, "Weather Forecasting Model using Artificial Neural Network", Procedia Technology, vol. 4, pp. 311-318, 2012.

[8] Augmenting Operating Systems With the GPU, 1st ed. Salt Lake city: University of Utah, School of Computing, 2012, pp. 1-5.

[9] M. Abadi et al., "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2015. [Online]. Available: http://download.Tensorflow.org/paper/whitepaper2015.pdf. Accessed: Mar. 3, 2017.

[10] N. Alham, M. Li, Y. Liu and S. Hammoud, "A MapReduce-based distributed SVM algorithm for automatic image annotation", 2017.

[11] D. Pop, "Title: Machine learning and cloud computing: Survey of distributed and SaaS solutions," 2016. [Online]. Available: https://arxiv.org/abs/1603.08767.

Accessed: Mar. 3, 2017.
[12] Nguyen, C. (2017). Arimo TensorSpark - SparkSummit. [online] Github. Available at: https://github.com/SparkTC/Spark-ref-architecture/blob/master/deep-learning/arimo/2016-02-Arimo-TensorSpark-SparkSummit.pdf [Accessed 4 Apr. 2017].
[13] A.Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware Resource Allocation Heuristics for Efficient Management of Data Centers for Cloud Computing," Future Generation Computer Systems, vol. 28, no. 5, pp. 755–768, May 2012.
[14] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick and D. Nikolopoulos, "Challenges and Opportunities in Edge Computing", *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, 2016.
[15] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The Cost of a Cloud: Research Problems in Data Center Networks," SIGCOMM Computer Communication Review, vol. 39, no. 1, pp. 68–73, Dec. 2008.
[16] A. Schumacher, "Hello, Tensorflow!", *O'Reilly Media*, 2017. [Online]. Available: https://www.oreilly.com/learning/hello-Tensorflow. [Accessed: 21- Apr- 2017].
[17] I. Kurochkin and A. Saevskiy, "BOINC Forks, Issues and Directions of Development1", *Procedia Computer Science*, vol. 101, pp. 369-378, 2016.