A Distributed Learning Simulation Platform for Edge Hierarchies

Alka Bhushan [*], Aniket Shirke [†], Govind Lahoti [‡], Umesh Bellur [§]
Indian Institute of Technology Bombay
Mumbai, India
Email: [*]abhushan@cse.iitb.ac.in, [†]anikets@cse.iitb.ac.in,[‡]govind14@cse.iitb.ac.in,[§]umesh@cse.iitb.ac.in

***Abstract*—We develop a distributed learning simulation platform that allows users to create multi-level Edge hierarchy for a given application by simulating resource constrained Edge devices and communication links amongst them. The resulting Edge computing hierarchy is used to run a given DNN for the application in a data distributed fashion, rolling up learned parameter values up a hierarchy of parameter servers that merge parameters received from the lower levels. The root of this hierarchy has the latest model, which is then pushed lazily back down the tree to the Edge servers. The platform can be used to study cost vs accuracy analysis of a given application for different Edge hierarchy configurations. We use handwritten digit recognition problem as a case study to show the usefulness of our platform.**

***Index Terms*—Edge Computing, Deep Neural Network, Simulation, Distributed Learning**

I. Introduction

The last decade has witnessed a surge in the amount of data derived from sensing devices deployed everywhere (smart phones, smart cameras etc.). Sensors are physically close to Edge data centers which house a relatively small amount of compute power. Deriving value from the sensed data usually implies training a learning model, such as an ANN or DNN situated in the Cloud (with a much larger amount of compute power compared with the Edge) where all the data is shipped. The Cloud then ships back the trained model to the Edge centers, which serves user requests based on this model. This traditional approach incurs a significant cost of data transfer and high latency for the applications that require large amount of training data and frequent updates in the model.

However, the computational power available at the Edge can be used in performing reasonably complex operations on-site instead of using the edge for only filtering data and sending suitably filtered data to the central Server/Cloud for training the model. Note that while a single Edge center may not be able to support very large models or a high rate of data, a set of Edge centers could cooperatively accomplish what the Cloud does. Learning at the Edge results in preserving privacy to users, reducing latency and communication bandwidth, and increasing robustness when connectivity to the Cloud is poor [3], [9]. This trending paradigm is referred by the term 'Edge Computing', where a part of the work happens at the Edge of the network which connects the physical world to the Cloud seamlessly. An Edge computing application uses the power of Edge devices in preprocessing and filtering the data [12], performing inference tasks [6], [10] and learning complex analytical models in a distributed setting [2], [4], [5], [8], [11].

There can be a multi-level hierarchy between cloud and sensing devices with edge devices of different configurations at each layer. These edge devices at multiple layers build a distributed computing hierarchy that can be scaled vertically as well as horizontally [10]. In such environments, the size of distributed computing hierarchy can also vary and a distributed computing hierarchy designed for an application may not be suitable for another application.

While there have been efforts to solve specific learning applications using distributed learning at the edge [2], [4], [5], [8], [10], [11], most of these efforts have presented a small sized single distributed configuration with one level hierarchy for evaluating the performance of distributed learning. In this one level of hierarchy, edge devices connected to sensors train the model using their local data and send the updates to the server for aggregation where a server can be cloud or an edge device. However, an application can be deployed by different organizations at different scale. Since communication cost cannot be same among geographically distributed edge devices, a single level hierarchy may not be an optimal distributed configuration for a large scale deployment. Therefore, levels in hierarchy may need to be increased vertically as well as horizontally to reduce the communication cost and the workload in case the application need to be deployed over a large area. The tradeoff to adding levels is a possible delay in the edge servers models reaching the same level of accuracy as that of a centralized cloud based learning system. Hence we need a tool to understand the tradeoff empirically.

In this work, we present a distributed simulation platform that can be used to simulate resource constrained Edge devices and sensors to build distributed computing hierarchy. The distributed computing hierarchy simulated by our platform can be used for testing the performance of different distributed setups of a given application. We demonstrate the use of our platform with an handwritten digit recognition problem and show how different distributed setup affects the accuracy and cost of the classification exercise.

The rest of the paper is organized as follows: Distribution hierarchical approach is described in Section II and Distributed computing simulation platform is presented in Section III. The case study is given in Section IV and related work is presented in Section V. Finally, the paper is concluded with the future

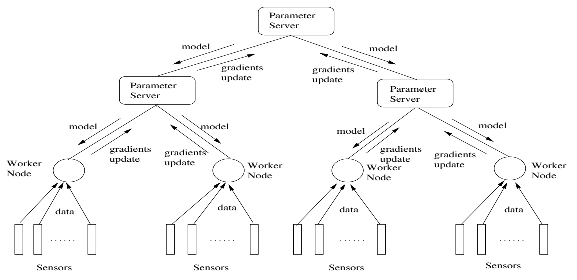

Fig. 1. Example of An Edge Hierarchy

work in Section VI.

II. Background

There are two approaches proposed in the literature for distributed training of a deep neural network [1]: model parallelism and data parallelism. The former typically splits each layer of the DNN to run on its own server while the latter trains different instances of the model with subsets of the data and then merges the learning from the instances using parameter server. Model parallelism is suited for very large models with many input features and data parallelism is suited for small models with large number of high speed data streams. In this work, we focus on the data parallelism approach where training data is distributed on multiple machines and assume that the complete model fits in any Edge device present in the computing framework.

A. Setup

Each edge center at the lowest level of hierarchy is attached to sensors that send data at a fixed data rate. We can abstract any data type from these sensors. Each edge center receives the data sent by these sensors during a time window where each time window has a fixed time duration and trains the model on this data. Periodically these edge centers send their model updates up to a parameter server which merges the learning of the different edge centers connected to it and sends the merged model back to its children. In order not to overwhelm a parameter server with too many Edge centers sending model parameters for merging, there are multiple parameter servers. The process of refining the model is incremental and hierarchical. Lowest level edge centers are the actual computational units, which train their model using the training data points, while parameter servers act as relay between the workers to communicate. When each parameter server merges gradients obtained from the lower level of hierarchy, it propagates the updates up the tree to higher level parameter servers. The root of the tree is where this process stops. The computation units at the lowest level act as worker nodes. We assume that the communication cost across each level in the hierarchy is not same. Communication cost increases as the data moves up the hierarchy. The setup of 4 worker nodes and 3 parameters servers with 3 level hierarchy is shown in Figure 1.

B. Learning Refinement in the Hierarchy of Nodes

Each worker node trains its own copy of the completed model using the data from current window and periodically sends the gradients to its parent. The parent node (parameter server) updates its model using the gradients obtained from the child nodes (parameter servers or worker nodes) by averaging the gradients obtained from each child. It then sends the gradients to its parent and so on. In the downward direction, every node obtains the latest copy of the model from its parent on demand. The averaging of gradients has shown to be useful in distributed configuration [1], [4], [8]. If each node sends its gradients to its parent after each update, the communication cost will be high which will increase with the increase in number of levels of hierarchy. To reduce the communication cost, we can use the trigger methods as defined below.

1) Time Based Trigger Method: As described in [1], [8] , time based triggers can be used to reduce the communication between edge devices across different levels of hierarchy. In time based method, an edge device waits for a fixed number of time units to send/fetch the updates to/from the parent edge device. Different time intervals are set across each level where frequent merging can be performed at the lowest level using short time interval.

2) Learning Based Trigger Method: In these types of methods, gradient updates are communicated upwards only if these updates help in converging the global model. We can identify these updates using one of the following approaches as given in [2], [11]: (i) Significance of gradient updates is computed at the worker nodes after learning from the current data and these updates are sent upwards if the significance is greater than a predefined threshold [2] (ii) Instead of significance, relevance of local updates are computed at the worker nodes with the global updates produced by all worker nodes. If the relevance of a local update is smaller than the threshold then update is discarded [11].

III. Simulator for Edge Hierarchies

Our simulator simulates scenarios such as link delays, streaming data and resource constrained edge devices, and provides an abstract class to implement an application.

A. Design

We use Kafka integration [1] to simulate sensors. Every sensor is assumed to have a sensor ID. Here, the publishing entity is a sensor, which publishes data to the topic which is the same as its own sensor ID. The sensor is effectively a python script which reads the sensor data from a file and dumps it at a fixed data rate into a Kafka server.

Docker containerization [2] is used to create resource constrained edge devices by containerizing code execution in the form of an image. Multiple Docker containers can be created as an instantiation of a single image. We consider computational capability and memory usage as resource constraints for an edge device.

[1]https://kafka.apache.org/
[2]https://www.docker.com/

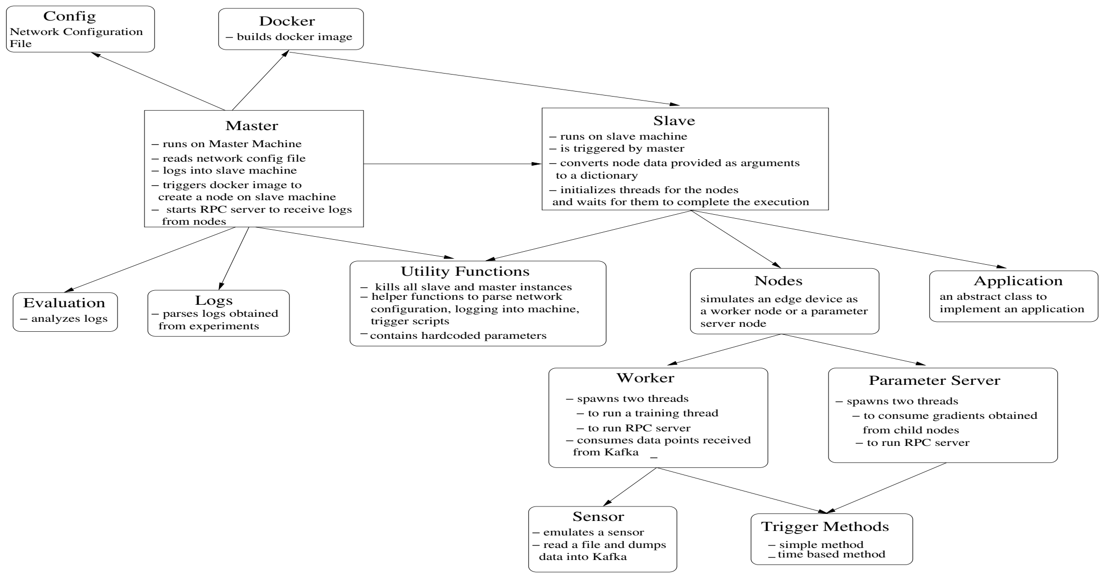
Config
Network Configuration File
Docker
– builds docker image
Master
– runs on Master Machine
– reads network config file
– logs into slave machine
– triggers docker image to create a node on slave machine
– starts RPC server to receive logs from nodes
Slave
– runs on slave machine
– is triggered by master
– converts node data provided as arguments to a dictionary
– initializes threads for the nodes and waits for them to complete the execution
Evaluation
– analyzes logs
Logs
– parses logs obtained from experiments
Utility Functions
– kills all slave and master instances
– helper functions to parse network configuration, logging into machine, trigger scripts
– contains hardcoded parameters
Nodes
simulates an edge device as a worker node or a parameter server node
Application
an abstract class to implement an application
Worker
– spawns two threads
– to run a training thread
– to run RPC server
– consumes data points received from Kafka
Parameter Server
– spawns two threads
– to consume gradients obtained from child nodes
– to run RPC server
Sensor
– emulates a sensor
– read a file and dumps data into Kafka
Trigger Methods
– simple method
– time based method

Fig. 2. Simulator Design

The design of our simulator is shown in Figure 2 and the source code is available on git [3]. Our Simulator follows Master-Slave model to automate execution of multiple experiments where Master is responsible for initiating the simulation of a scenario provided by a configuration file (yaml file) and Slave is responsible for creating and executing the scenario received from the Master. The Master and Slave are python scripts. The Master program runs on master machine. It does the following: (i) reads configuration file provided by an user and logs into slave machine, (ii) triggers docker image to create nodes on the slave machine, (ii) starts RPC server to receive logs from the nodes. It uses helper functions provided in utility functions to parse network configuration file, logging into machine and trigger scripts. Utility function also contains hard-coded parameters such as RPC server port, data rate, Kafka server address, trigger commands, constants for logging and reporting, and communication protocol between parameter server and child node. All master and slave instances are killed using scripts provided in utility functions.

The slave program runs on slave machine but is triggered by master program from master machine as it has dependencies of RPC server (for reporting) and the parent-child hierarchy defined by the configuration file. It converts node data provided as arguments to a dictionary and initializes threads for the nodes and waits for them to complete the execution. A node is a simulated edge device which acts as either a parameter server or a worker. Each worker node spawns two threads where one thread is used in training the model and pull/push the current model from the parent and the second thread is used for inter-node communication. Similarly, each parameter server node spawns two threads where one thread is used in consuming the gradients from the child nodes and the second thread is used for inter-node communication. A worker node is connected to Kafka server and consumes data points in batches received from Kafka. Slave program uses application to run on the edge hierarchy. An abstract class to implement an application is provided. We have implemented simple and time based trigger methods in the Simulator. In simple method, model and gradient updates are communicated each time. Time based trigger method is same as described in Section II-B1.

A configuration file is an yaml file that is provided by an user. A sample of configuration file is given in Figure 3. It contains the following information:

1) common fields for the hierarchy such as bandwidth between nodes, default bandwidth, time interval for a window, window limit, address of kafka server and directory of test data.
2) docker specific fields such as CPU resources and memory to be allocated to container and docker image to be used.
3) fields for configuring nodes such as node id, port, test directory , time interval for a window and window limit.
4) credentials of host machine on which nodes will run as containers
5) arguments for application such as input size, output size, number of layers, size of each layer, learning rate, batch size and number of epochs.
6) trigger method used for communication between parent and child node.

Each node sends log to master machine. The structure of the log is as follows:

- `NODE_ID`: ID of the node which reported the log
- `TYPE`: There are three types of logs:
  - `CONNECTION`: Log indicates the communication of the node with any other entity: parent, Master or Kafka server

[3]https://github.com/govindlahoti/TreeNN

```
## Bandwidth available to nodes
bandwidths:
  - src_id: 1
    dest_id: 2
    bandwidth: 1000000
default_bandwidth: 1000000
## Data streaming info
default_window_interval: 60
default_window_limit: 20
default_kafka_server: "###"
default_test_directory: "/data/"
## Docker specific fields
default_cpus: 0.8
default_memory: "256M"
default_docker_image: "image_name"
default_host_test_directory: "~/mnist_data/test"
default_policy: 0
default_args: 0
## Machine info
machine:
-   ip: "###"
    username: "###"
    password: "###"
## arguments to be passed to the learning model
application_arguments:
-   model            : 'MNIST'
    input_size       : 784
    output_size      : 10
    num_hidden_layers : 3
    hidden_layer_sizes: [20,10,10]
    alpha            : 0.4
    batch_size       : 20
    epochs           : 10
## Policy for sharing models
policy:
-   type: 'SimplePolicy'
    args:
-   type: 'TimePolicy'
    args:
        push_interval: 120
        pull_interval: 120
## Edge hierarchy
nodes:
-   id: 1
    port: 8000
    machine: 0
    cpus: 2
-   id: 2
    port: 8006
    machine: 0
    parent_id: 1
    sensors: [0]
## cloud configuration
cloud:
    machine: 0
    port: 9000
    cpus: 2
    memory: "500M"
```

Fig. 3. A Sample Configuration File

   - `STATISTIC`: Log contains the following statistics about the simulation: window id, run time, CPU time and memory usage for processing whole data in a window and accuracy of the model computed using test data.
   - `DONE`: Special type reserved to indicate that the node has finished simulating
- `PAYLOAD`: Payload depends on the type of log.
- `TIMESTAMP`: Timestamp of the log obtained by calling `time.time()` in python

A parsing script is provided to parse the logs to extract the required information for visualizing the results.

B. Features

Our Simulator has the following features:

1) It allows simulation of edge nodes of different capacities
2) Users can specify different trigger methods for sending model parameters up the hierarchy, and
3) It allows simulation of different network delays and bandwidths for communication links.
4) It provides an abstract class to build any supervised learning application.

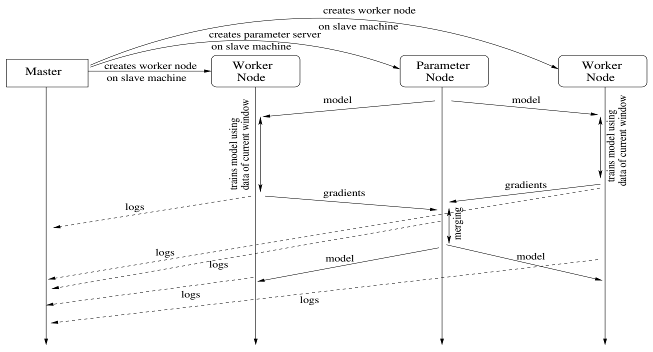

Fig. 4. Simulator Run with two worker nodes and one parameter server

C. A Run of The Simulator

A simulator run with two worker nodes and one parameter server is shown in Figure 4. Initially, a Docker image is created which contains the script for running edge nodes. The node ID, node data (from the configuration file) and the address of the Kafka server is passed as an environment variable. The Master now logs into the Slave machine, specifies the number of cores and memory to be given to each edge node and triggers a Docker container. Nodes of the Edge hierarchy as given in the configuration file are built. The learning on the Edge hierarchy starts of by initializing the learning model at the worker nodes with the same parameter values across all worker nodes. The learning moves up the hierarchy as learning progresses. Each node records the window id, run time, CPU time, memory usage and accuracy in a log and sends it to the master. A Worker node finishes its learning after processing a specified number of windows of data where a window contains data points arrived within a specified time interval. For a given internal node, the node stops merging after all children stops running. Thus, the learning starts and ends in a bottom-up manner.

D. Performance Metrics

We use the following metrics to compare different configurations:

1) Model Performance:

a) **Accuracy::** Each worker node learns the model using the local data available to it from sensors. We measure the accuracy of the model at each worker node using the test data.

b) **Latency:** : Latency is the maximum time required by a worker node to receive a model updated by all worker nodes. Let t^i_j be the timestamp when j^{th} time window is processed at worker node i and tr^i_j be smallest timestamp when worker node i receives the model from its parent which is updated using time window j or greater. For each worker node i, time lag at window j is latency$^i_j = min_{k\in[1,n]}(tr^i_j - t^k_j)$ and latency is latency$^i = max_{j\in[1,T]}(latency^i_j)$. Thus, Latency of the given configuration is $max_{i\in[1,n]}(latency^i)$.

c) **Accuracy Lag::** Each worker node learns the model using the local data seen so far from sensors. It is expected that the worker node can answer queries with high accuracy on data seen by it instead of queries on data seen by other worker nodes because it will take some time to transfer the learning between two distant worker nodes. Thus, at any given point of time, there will be a delay in merging the learning of all the data points seen by all worker nodes. In order to compare the difference between distributed learning and centralized learning, this metric is important as the difference between the accuracy of the individual worker node and that of the cloud is captured. If $A_i(t)$ is the Accuracy of i^{th} node and $A(t)$ is the Accuracy of the Cloud at time t, then AccuracyLag$(t) = A(t) - A_i(t)$

2) Computational Performance: For a given hierarchical configuration, trigger methods and learning model, we measure the total CPU cost and memory usage of each node, and network cost for each link in the hierarchy.

IV. Case-Study: Handwritten Digit Recognition Problem

We use Handwritten Digit Recognition Problem as a case study to show the usefulness of our simulator. The problem is to classify the biased data stream of handwritten digits where each stream is biased towards a specific digit.

A. Model

We have used a simple Deep Neural Network (DNN) with 3 hidden layers. Every image consists of 784 pixels and the input to the Neural Network is a 784×1 feature vector. The output of the DNN is a 10×1 vector, each denoting the probability of the input image being 0 to 9 respectively. The neural network has three hidden layers of sizes 20,10 and 10 respectively. The number of epochs is 10 and batch size is 20. The learning rate is 0.4.

B. Dataset

We use the Modified National Institute of Standards and Technology (MNIST) dataset [4] which consists of 60000 training images of handwritten digits and 10000 test images. We created 10 data streams (corresponding to digit 0-9) which are biased towards a specific digit x, where x appears 64 % of the times in the stream and the rest of the digits occupy 4 % of the stream. The aim is to recognize a given input digit with maximum accuracy.

C. Experiments

We performed our experiments on a virtual machine with 2.10 GHz 16 core Intel(R) Xeon(R) E5-2683 v4 processor and 32GB of main memory. Two different types of experiments are performed to show the usefulness of our simulator. The first set of experiments considers two level hierarchy which contains only one parameter server connected to all worker nodes. In the second set of experiments, we consider two and three level hierarchy with 6 worker nodes. For both the experiments, each worker node in the configuration is assigned 0.8 core and 256 MB main memory. Root node in each hierarchy is assigned 2 cores and 256 MB main memory and each parameter server other than root node is assigned 1 core and 256 MB main memory. For cloud model, a container of 10 cores and 512 MB main memory is created to run the application. In the cloud model, all sensors send the data to the cloud and DNN model is trained in the cloud using the data received from the current time window. We have used simple merging method where model updates are sent to parent nodes after processing of each time window. The size of the window is considered 60 seconds for each experiment and data rate of each stream is 4 images/second. After processing 20 windows, each worker node finishes its learning and root node stops merging after all its children stops running. Each experiment reports a log file. Results of both types of experiments are given next.

1) Experiment 1: Varying number of worker nodes: These experiments show the variation in accuracy and accuracy lag with the increase in number of worker nodes from 3 to 7 while keeping same time window, data rate and edge node configurations.

a) **One Parameter Server - Three Workers:** A single parameter server (Node 1) is connected to three worker nodes (Node 2, Node 3 and Node 4). The i$^{\text{th}}$ worker node (i=2,3,4) receives a biased data stream of digit **d** if and only if $d\%3 = i-2$. The run time of each worker node is 20 minutes and CPU process time of each worker node is 11 minutes, 8 minutes and 8 minutes respectively. The communication cost between parameter server and each worker node is 4.9 MB.

The model accuracy increases for all the nodes with the increase in time as more data is seen by worker nodes and saturates after a certain time. As show in Figure 5, Node 1 initially does not perform as well as the Cloud, but eventually the model accuracy of Node 1 and the Cloud are comparable. The worker nodes perform very well in classifying the digits they received from their own data stream (more than 80% accuracy). Even for the data not received locally, the nodes do well in classification, as each worker node has an accuracy of approximately 65% in classifying other digits. The digits seen by worker nodes are distinguished by colors in Figure 5. As we can see in the Figure, the model converges approximately in 600 seconds.

As shown in Figure 6, the accuracy lag of some digits at root node 1 is negative initially as cloud accuracy may not be as good as the accuracy in distributed learning but reaches close to 0 eventually, implying that the distributed learning model is on par with the centralized learning model. We notice that the accuracy lag of digits 0,3 and 9 at node 2 is negative as worker nodes will perform better than the Cloud on classifying data points sampled from their local data streams.

Latency of this configuration is approximately 45 seconds.

b) **One Parameter Server - Five Workers:** A single parameter server (Node 1) is connected to five worker nodes (Nodes 2 to 6). The i$^{\text{th}}$ worker node (i=2,3,4,5,6) receives a biased data stream of digit **d** if and only if $d\%5 = i - 2$. The experiment ran for 20 minutes and CPU process time of each worker node is 5 minutes.

The communication cost from parameter server to each worker node is 4.9 MB. Note that the communication cost

[4]http://yann.lecun.com/exdb/mnist/

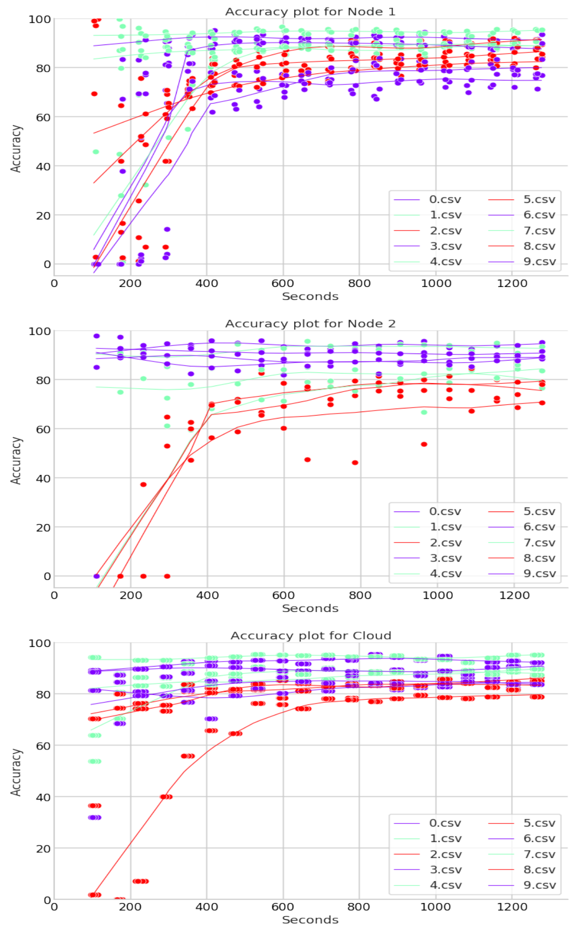

Fig. 5. Model Accuracy for MNIST. 1 Parameter Server - 3 Workers

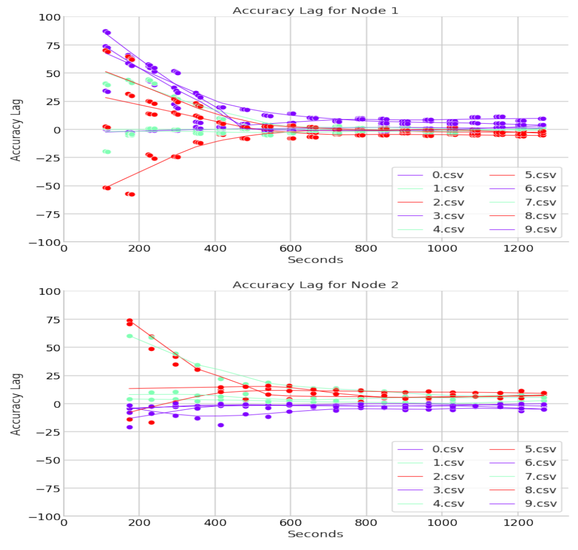

Fig. 6. Accuracy Lag for MNIST. 1 Parameter Server - 3 Workers

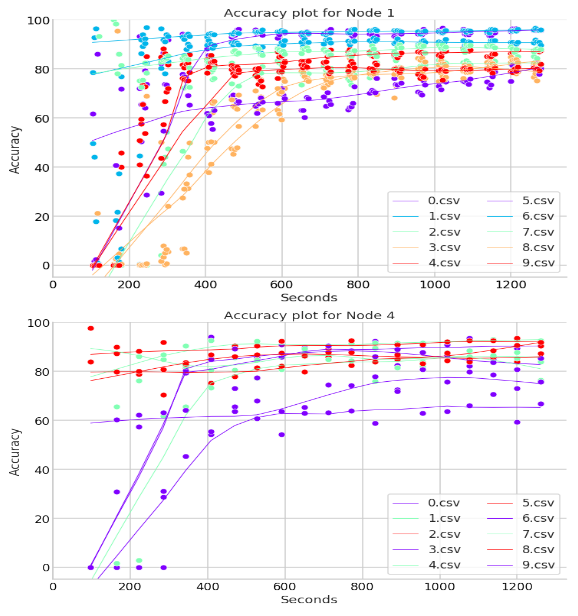

Fig. 7. Model Accuracy for MNIST. 1 Parameter Server - 5 Workers

remains same as in 1 parameter server and 3 worker nodes because total number of windows processed at each worker node remains same i.e. 20 and model updates are communicated after each time window. Since number of worker nodes are increased from 3 to 5. The total number of images per window reduces as only 2 sensors are connected to each worker node instead of 3 sensors per worker node. Due to less number of images processed in each time window, the accuracy of the model was not as good as with 3 worker nodes. The accuracy of the root node as shown in Figure 7 is 75 % after converging the model which is 5 % less than the accuracy of the cloud model. In Figure, Node 1 is a root node and node 4 is a worker node. Due to lack of space, the result of cloud accuracy is not shown here. Also, there is a delay in converging the model i.e. 1000 seconds. As shown in Figure 8, initially accuracy lag is more than the accuracy lag achieved with 3 worker nodes, but reaches zero towards the end. The latency is approximately 51 seconds.

c) **One Parameter Server - Seven Workers:** A single parameter server (Node 1) is connected to seven worker nodes (Nodes 2 to 8). The i$^{\text{th}}$ worker node (i=2,3,4,5,6,7,8) receives a biased data stream of digit **d** if and only if $d\%7 = i - 2$. The experiments ran for 19 minutes. The CPU process time is 5 minutes for first three worker nodes and 3 minutes for remaining 4 worker nodes. Worker nodes 2, 3 and 4 receives data of 2 digits and remaining sensors get data of 1 digit only. Since time window and data rate remains same as used with 3 and 5 worker nodes, the number of images per window is smaller than the number of images per window for 3 and 5 worker nodes. In this case, as shown in Figure 9, the model started converging towards the end of 20 windows. The accuracy was not as good as with 3 and 5 worker nodes. As expected the accuracy lag is also not approaching to zero as shown in Figure 10. These results show that the 20 time windows are not enough for the current data rate and time window interval. The latency is approximately 59 seconds.

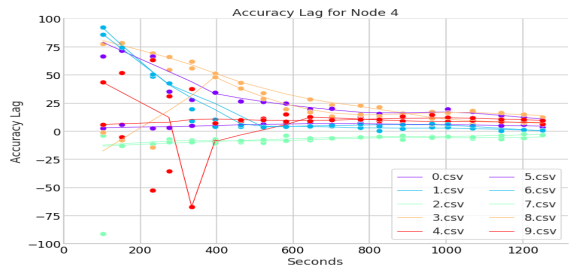

Fig. 8. Accuracy Lag for MNIST. 1 Parameter Server - 5 Workers

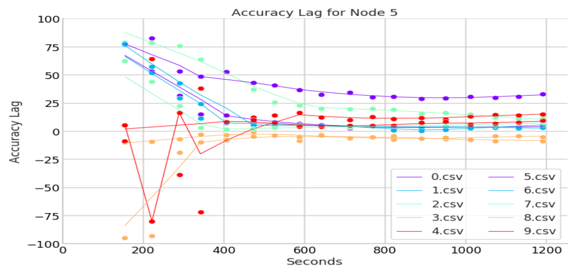

Fig. 10. Accuracy Lag for MNIST. 1 Parameter Server - 7 Workers

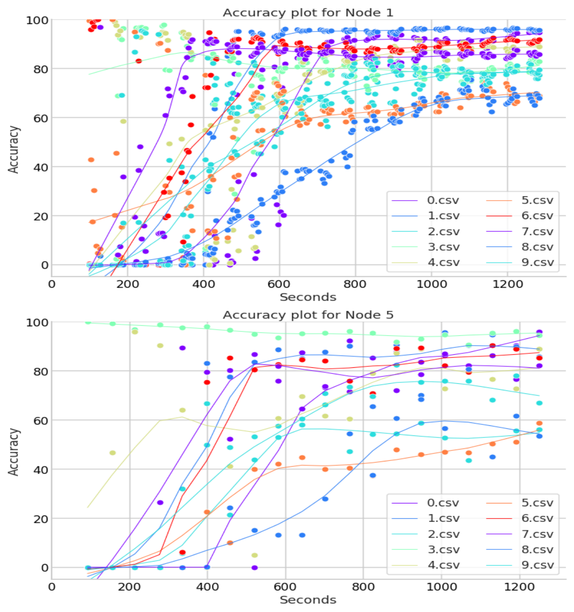

Fig. 9. Model Accuracy for MNIST. 1 Parameter Server - 7 Workers

2) Experiment 2: Multilevel hierarchy:

a) **One Parameter Server - Six Workers:** A single parameter server (Node 1) is connected to six worker nodes (Nodes 2 to 7). The i$^{\text{th}}$ worker node (i=2,3,4,5,6,7) receives a biased data stream of digit **d** if and only if $d\%6 = i - 2$. The experiment ran for 19 minutes. The CPU process time of first 4 worker nodes is 5 minutes and next two worker nodes is 3 minutes. The communication cost between each link is 4.9MB and the model is converged approximately at 1200 seconds. Each worker node classifies own digits with more than 80 % accuracy and other digits with more than 60 % accuracy. However, root node classifies digits with at least 75% accuracy which is 5 % less than the cloud accuracy. The latency of the configuration is approximately 57 seconds.

b) **Three Parameter Servers - Six Workers:** In this configuration, two level hierarchy is built where one parameter server is at level 2 and two parameters are at level 1. The worker nodes are at level 0 and each parameter server of level 1 is connected to 3 worker nodes. The i$^{\text{th}}$ worker node (i=4,5,6,7,8,9) receives a biased data stream of digit **d** if and only if $d\%6 = i-4$. The experiment run time and CPU process time is same as with one parameter and six worker node. The communication cost between each worker node and its parent node is same as 4.9 MB. However, the communication cost between root node and its child parameter nodes is 9.6 MB and 10.8 MB respectively. Note that the total communication cost between root node and its children is 20.4 MB which is less than the total communication cost between root node and its child worker nodes in case of one level hierarchy. The figure 11 shows the accuracy of root parameter server, parameter server at level 1 and a worker node. Each worker node classifies own digits with more than 80 % accuracy but the classification accuracy of other digits is not as good as with one level hierarchy. However, classification accuracy of root node is comparable to classification accuracy of cloud and is more than the classification accuracy of root node with one level hierarchy. But, the latency is comparable.

3) Discussion: From these experiments we observed that as the number of worker nodes increases, classification accuracy decreases and the model convergence time increases when data rate and time window remain same. Further, we observed that when we increase the number of levels in hierarchy, the communication cost at top level in hierarchy reduces with some loss in accuracy at worker nodes.

V. Related Work

Till date, following two different types of problems on distributed Edge hierarchy have been investigated in the literature. First work on Distributed deep neural network over Edge hierarchy is presented in [10] for answering classification queries at much lower cost than answering the queries using cloud based solution. The training of the model happens on a single powerful server. The sections of the trained model are mapped onto distributed Edge hierarchy. The classification queries are inferenced by aggregating the output obtained from each end device. Another work on distributed Edge hierarchy is on approximate computing of analytical queries where incoming high speed data from the sensing devices is filtered at each level of hierarchy while going up to the root [12]. Different types of analytical queries can be asked from this sampled data. The accuracy of the queries depends upon the sample size used at each Edge device in the hierarchy where sample size depends upon the budget constraints given by an user.

Orthogonal to this, a Geo-distributed machine learning system, Gaia is proposed to efficiently run machine learning

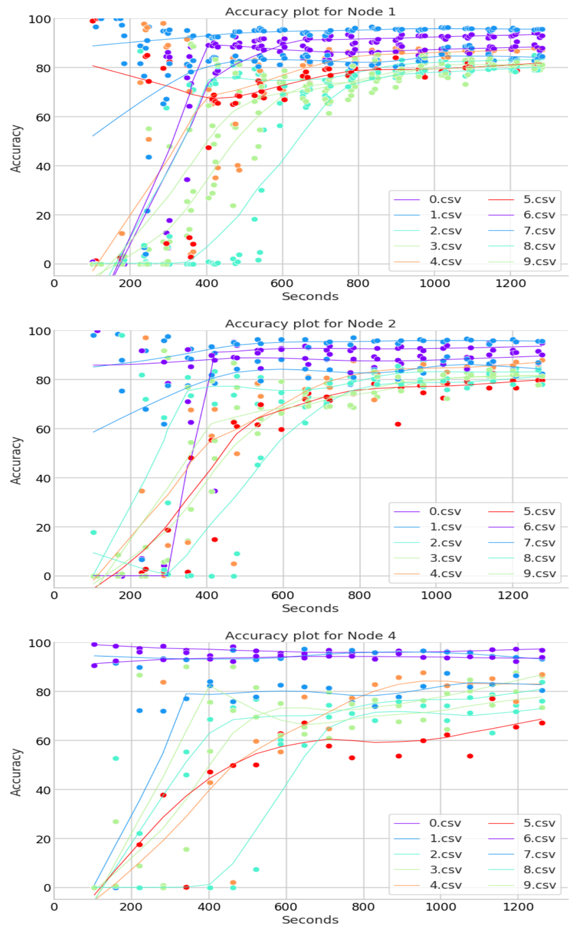

Fig. 11. Model Accuracy for MNIST. 3 Parameter Servers - six Workers

algorithms on data generated across the world [2]. Gaia reduces the communication cost across data centers by excluding non-relevant updates from data transfer. In this system, worker nodes can communicate to all parameter servers but with different communication cost and worker nodes are powerful data centers. Further, Distributed learning on mobile devices has been investigated in [8], [11] where mobile devices were used as computing devices. A global model is stored in a central server which is updated synchronously by selecting few mobile devices at one time. The frequency of updates is kept very low due to large communication cost. Recently, distributed learning on resource constrained Edge devices is investigated where convergence rate of gradient descent method from a theoretical point of view is analyzed and synchronized training method is used [5]. These methods use only two level of hierarchy and use different triggering methods to reduce the communication cost.

Other than this, edge computing platforms have also been introduced to provide computing and storage resources at the edge for creating different types of services [7]. The platform has been used to build EdgeEye framework for video analysis [6]. The framework provides API to transform trained DNN models to deployable components on edge.

VI. Conclusion

In this work, we developed a distributed learning simulation platform that allows users to simulate resource constrained edge devices and build hierarchy using these devices by creating communication links amongst them. The resulting hierarchy was used to run a DNN model in a data distribution fashion. Our simulation platform is capable of performing cost vs accuracy analysis for a given application. The platform was tested for a case study on handwritten digit recognition problem. This is a preliminary work done in simulating an Edge computing environment for measuring the cost of the framework. The results showed that distributed configurations affect the performance of the model. The work is in progress and we are currently working on learning based triggering methods and multiple levels of hierarchy into the platform as well as experimenting with other applications.

Acknowledgement

The authors would like to thank Mayank Singhal for generating the results.

References

[1] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1223–1231, 2012.

[2] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu. Gaia: Geo-distributed machine learning approaching lan speeds. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 629–647, 2017.

[3] G. Kamath, P. Agnihotri, M. Valero, K. Sarker, and W. Z. Song. Pushing analytics to the edge. In *IEEE Global Communications Conference*, GLOBECOM, pages 1–6, Washington, DC, 2016.

[4] J. Konečný, H. McMahan, D. Ramage, and P. Richtárik. Federated optimization: Distributed machine learning for on-device intelligence, 2016. arXiv:1610.02527 [cs.LG].

[5] K. Leung, S. Wang, T. Tuor, T. Salonidis, C. Makaya, T. He, and K. Chan. When edge meets learning: adaptive control for resource-constrained distributed machine learning. In *IEEE INFOCOM*, 2018.

[6] P. Liu, B. Qi, and S. Banerjee. Edgeeye: An edge service framework for real-time intelligent video analytics. In *Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking*. ACM, 2018.

[7] P. Liu, D. Willis, and S. Banerjee. Paradrop: Enabling lightweight multi-tenancy at the network's extreme edge. In *IEEE/ACM Symposium on Edge Computing (SEC)*, 2016.

[8] H. McMahan, E. Moore, D. Ramage, and B. y. Arcas. Federated learning of deep networks using model averaging, 2016. arXiv:1602.05629v1 [cs.LG].

[9] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[10] S. Teerapittayanon, B. McDanel, and H. T. Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *IEEE 37th International Conference on Distributed Computing Systems*, ICDCS, pages 328–339, Atlanta, GA, 2017.

[11] L. Wang, W. Wang, and B. Li. Cmfl: Mitigating communication overhead for federated learning. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems*, ICDCS'19, 2019.

[12] Z. Wen, D. Quoc, P. Bhatotia, R. Chen, and M. Lee. Approxiot: Approximate analytics for edge computing. In *IEEE 38th International Conference on Distributed Computing Systems*, ICDCS'18, pages 411–421, 2018.