Distributed Learning on Edge for IoT A Simulator Based Approach

Aniket Shirke(150100012)
Guide: Prof. Umesh Bellur

November 24, 2018

Abstract

Emerging technologies and applications in Internet of Things (IoT) generate large amounts of data at the network edge. Machine learning models are often built from the collected data, to enable the detection, classification, and prediction of future events. Due to bandwidth, storage, and privacy concerns, it is often impractical to send all the data to a centralized location like Cloud. Hence it becomes imperative to learn the machine learning models on the Edge in a distributed fashion. We propose a simulator-based approach to simulate the learning of such models on the Edge.

Contents

1 Introduction

There has been an explosion in the deployment of IoT devices, generating enormous amounts of data in the form of fast-moving data streams. Typically this data is pushed up to a central server where either a classification or a prediction model can be trained and this method is not suitable for handling enormous amounts of data. Another concurrent trend has been the emergence of the Edge, having limited computational resources, as an alternative to reliance on Cloud resources. Rather than sending all data up to the Cloud, we can have a hierarchy of Edge centers that will train the model in a data distributed manner and push the model up to the Cloud.

In data distributed learning of models, different machines have a complete copy of the model. Each machine simply gets a different portion of the data and results from each are combined using some logic guaranteeing the convergence. We will follow a similar approach, where each Edge device will receive data from different sensors and maintain a complete local copy of the model.

Although we are distributing the load of incoming data among the Edge devices for data distributed learning of a model, there is a network cost of exchanging model parameters during the training phase. Another major factor is the accuracy of the model in the Cloud - as we are learning the model locally and pushing it up incrementally, there is an accuracy lag which will be experienced as any local changes in the model will not be reflected immediately in the Cloud.

To obtain the cost vs accuracy analysis of a given model, we need a simulation environment where we can simulate different degrees of distribution and their effect on the cost of data transfer as well as on the accuracy of the model at Edge devices as learning progresses. This will help us decide on a distribution strategy of the Edge devices. The objective of the Simulator: For a given data stream and available Edge devices with known computational constraints, compare the performance of a given supervised learning algorithm for multiple logical trees of Edge devices and suggest an optimal logical tree for that deep learning model.

The proceeding sections in the report describe functionality and application of the Simulator, some important technical features and the experimental evaluations which are to be done.

2 Related Work

[1] presents an extensive survey on Deep Learning for Big IoT Data and Streaming Analytics in which they review different applications in various sectors of IoT.

[2] present a novel Distributed Deep Neural Network(DDNN) framework and an implementation that maps sections of a Deep Neural Network onto a distributed computing hierarchy. Their application is specific to the image processing domain and hence all the layers in DDNN are binary.

[3] analyzed the convergence bound for distributed learning with non-i.i.d. data distributions. Using this theoretical bound, they propose an algorithm in order to minimize the loss function under a resource budget constraint. They make an extensive analysis on the convergence rate of distributed gradient descent from a theoretical point of view by assuming a given resource constraint budget.

3 Air Quality Prediction problem

This section is borrowed from Chapter 4 of [4]

3.1 Introduction

Air Quality in metro cities across the world is deteriorating day by day due to pollution. Hence it is essential to find the air quality in a specific region of interest. Air Quality can be predicted using data from various meteorological and AQI sensors deployed across the metropolitan. Traditionally the data from these sensors used to be sent to the Cloud for training the machine learning models for AQI prediction. We intend to do same analytics using Edge devices. We define the problem statement more formally in the next section.

3.2 Problem Statement

Given a geographic area divided into a grid of cells [g_1,g_2,g_3,...g_n] of equal size where each grid cell contains sensors to collect weather information and an Edge device for local computation and communication with edge devices in the neighborhood, and AQI sensors in few grid cells, schedule the AQI queries on edge devices and answer these queries such that computational and network costs are minimized.

Assumptions:

- Each grid cell contains sensors to get the meteorological data. These sensors gives following information at regular time interval: Weather, Temperature, Pressure, Humidity, Wind Speed, Wind Direction.

- Very few grid cells have sensors to collect air quality data. These sensors collect following information at regular interval: $PM_{2.5}$, PM_{10}, NO_2, CO, O_3, SO_2.
- Each grid cell has one edge device.

The raw data collected from these sensors need to be preprocessed because all sensors may not provide data exactly at the same time. Some values may also be missing. Moreover, there may be multiple sensors to collect weather data. In such case, aggregation and interpolation is required. Some locations may be queried many times while some locations may rarely be queried. In such cases, queries on edge devices will not be equally distributed. Some edge devices may face many queries and some edge devices may not face any query. Some queries may be repeated again frequently.

To answer a query, the following information is required:

- Weather data of the queried grid cell, road network and POI information from all the grid cells in its neighborhood.
- AQI values, if grid cell has AQI sensor. Otherwise, AQI values from neighborhood grid cells where AQI sensors are located.

The collected data is fed into the learned model to compute AQI value. We assume that each edge device has the model for its own grid cell and has capability to communicate with devices in the neighborhood. In such case, if query is given to an edge device which does not belong to the queried grid cell then extra cost is required for sending the model and all the data to compute AQI values.

As mentioned before, the model is traditionally learnt in the Cloud. We would like to explore if the same can be done using the federation of edge devices.

3.3 Algorithm

We use Downpour Stochastic Gradient Descent algorithm for sharing the models/gradients up and down the levels in the tree.

Algorithm 1.1: DOWNPOURSGDCLIENT($\alpha, n_{fetch}, n_{push}$)

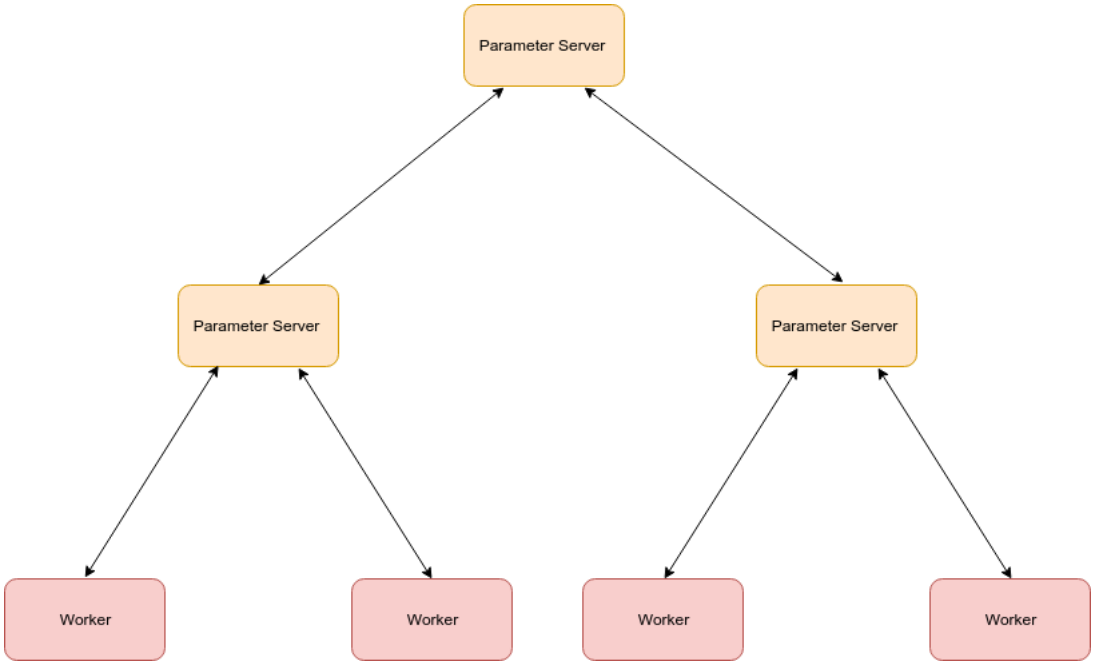
```
procedure StartAsynchronouslyFetchingParameters(parameters)
  parameters ← GetParametersFromParamServer()

procedure StartAsynchronouslyPushingGradients(accruedgradients)
  SendGradientsToParamServer(accruedgradients)
  accruedgradients ← 0

main
  global parameters, accruedgradients
  step ← 0
  accruedgradients ← 0
  while true
      do { if (step mod n_fetch) == 0
             then StartAsynchronouslyFetchingParameters(parameters)
           data ← GetNextMinibatch()
           gradient ← ComputeGradient(parameters, data)
           accruedgradients ← accruedgradients + gradient
           parameters ← parameters − α ∗ gradient
           if (step mod n_push) == 0
             then StartAsynchronouslyPushingGradients(accruedgradients)
           step ← step + 1
```

4 Simulator Functionality

Each Edge device at the lowest level, which we will refer to as a Worker, receives a subset of data closest to it and trains the model on the data it gets. Periodically these Worker nodes send their model parameters up to a Parameter Server (hosted at an Edge device) which merges the learning of the different Edge devices and sends the merged model back to the lower level edge devices, i.e. Workers. In order not to overwhelm a Parameter Server with too many Worker nodes, there can be possibly multiple parameter servers for merging model parameters. The process of refining the model is therefore incremental and hierarchical.

The Simulator treats the Parameter Server and the Worker as nodes in the learning tree. The Simulator starts of by initializing the root node of the learning tree first and then starting the corresponding children. A Worker node finishes its simulation after processing a specified number of windows of data. For a given internal node, the node stops simulating only after all children finish simulating. Thus, the Simulation starts in a top-down fashion and ends in a bottom-up manner with respect to the learning tree

We now move on to the current functionality the Simulator performs to learn a given model.



4.1 Worker

The leaves of the learning tree consists of worker nodes. A Worker node spawns two threads:

- To run the training thread. For each window of data the Worker node processes, it:
  1. Pulls the model from parent
  2. Runs training algorithm as defined by the application
  3. Logs Statistics for the window
  4. Pushes new model to the parent
- To run the RPC server for inter-node communication

The following statistic is recorded by the Worker node after processing every window:

- Window ID: ID of the window of data processed
- Run time: Total time taken to process the window, calculated using `time.perf_counter()` in python

- Process time: CPU time taken to process the window, calculated using `time.process_time()` in python
- Memory usage: Memory usage recorded at the end of processing the window
- Accuracy: To analyse the spatial and temporal variation in accuracy, the model is evaluated against different test sensor data

4.2 Parameter Server

All the internal nodes of the learning tree are Parameter servers, which maintain the parameters of the model and do the job of merging models received from child Worker nodes. In case of supervised learning, the gradients are parameters which are consumed by the Parameter Server. A Parameter Server spawns two threads:

- To consume the gradients being obtained from child Worker nodes. For each merging of gradients, it:
    1. Monitors the queue of acquired gradients from the child Worker nodes
    2. Pulls model from parent
    3. Logs the accuracy of the model before merging gradients
    4. Modifies its own model using those gradients
    5. Logs the accuracy of the model after merging gradients
    6. Pushes new model to the parent (if it is not the root node)
- To run the RPC server for inter-node communication

4.3 Application abstraction

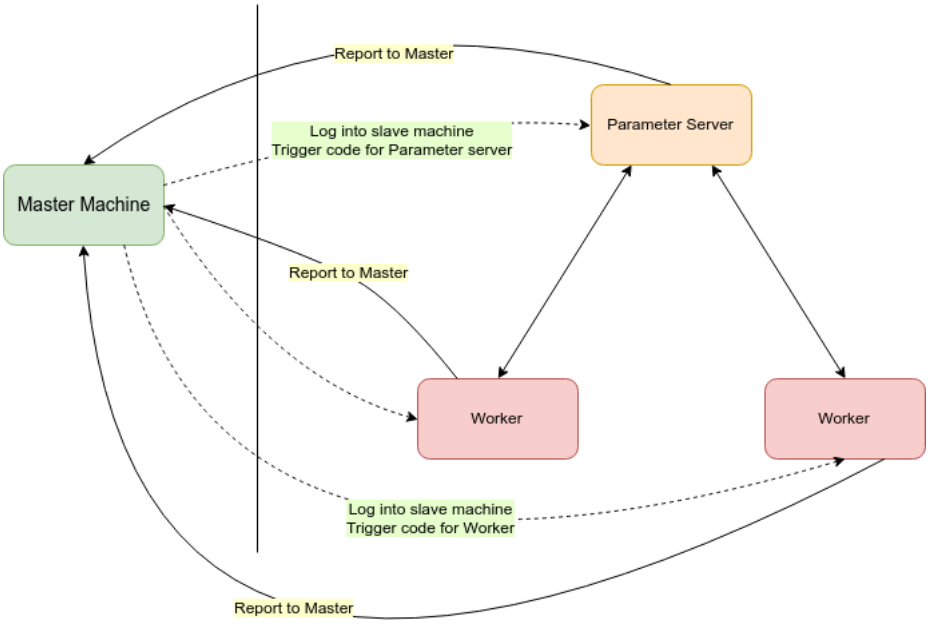
Even though the application under consideration is Air Quality Index prediction, the Simulator can be generalized to simulate the class of supervised learning problems. One needs to implement the abstract class Application to simulate another supervised learning application.

```
class Application(AbstractBaseClass):

        #------------------ For Parameter Servers -------------------
        def get_model(self):
        """
        Returns the current model.
        Used by Parameter servers for merging gradients and biases
        For supervised learning, the weights and biases are returned as
        they are sufficient in representing the model: Input*w + b = Output
        """

        def apply_kid_gradient(self, weight_gradient, bias_gradient):
        """
        Update the model (weights, biases) by adding the gradients
        obtained from the child node
        """

        #------------------ For Worker nodes ------------------------
        def train(self,training_data, *args):
        """
        Method which defines the training algorithm used by the application.
        In case of AQI, the algorithm is Stochastic Gradient Descent
        """

        #------------------ For Both --------------------------------
        def use_parent_model(self, weights, biases):
        """
        Fetch model from parent node
        """

        def get_and_reset_acquired_gradients(self):
        """
        Returns the acquired gradients in the model. Also resets them to zero
        """

        def evaluate(self,test_data):
        """
        Evaluate the current model using test_data
        """
```

5 Simulator Features

To simulate real life scenarios, issues such as link delays, streaming data and resource constraints have been incorporated in the Simulator. This section elaborates on the Master Slave model - to automate execution of multiple experiments, Kafka integration for streaming data and Docker Containerization.

5.1 Master Slave Model



5.1.1 Motivation

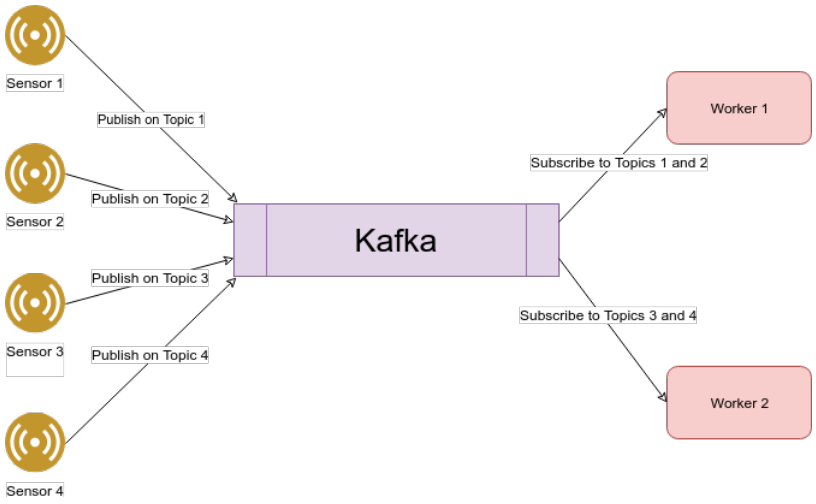
Training Deep neural networks for the AQI problem takes considerable amount of time, sometimes even a few hours. In order to ease the execution of experiments, a Master-Slave model is proposed so that different experiments can be scheduled on a Master machine to collect simulation logs submitted by Slave machines.

5.1.2 Design

The Master and Slave machines are Linux servers. The Master starts an RPC(Remote Procedure Call) server to receive logs about the simulation from the Slaves. It is assumed that the code required for running the nodes is present on the Slave machine.

The Master reads the credentials of the Slaves from the configuration file, logs into the Slave machines and triggers nodes with the option to containerize the node.

5.2 Kafka Integration



5.2.1 Motivation

In real world scenario, there are sensors that stream data to the Cloud after regular interval. It is expected that the Worker nodes running on the Edge receive data from sensors. To simulate this effect of streaming data from sensors on training the model, Kafka has been used to leverage its publish-subscribe model.

5.2.2 Design

In the publish-subscribe messaging paradigm, publishers publish messages to a specific topic and subscribers consume messages related to the topic by subscribing to that topic.

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.

In the configuration file, an array of sensors is assigned to each Worker node. Every Worker node then subscribes to the corresponding sensor ID and consumes data for training the model.

5.3 Containerization

5.3.1 Motivation

Edge devices have constraints on their resources, in terms of computational capability and memory usage[6]. In order to simulate these effects, Docker has been used to impose resource constraints by containerizing code execution.

5.3.2 Design

Docker allows us to containerize our code in the form of an image, and multiple Docker containers can be created as an instantiation of a single image. Along these lines, a Docker image is initially created which contains the script for running Edge nodes. The node ID, node data (from the configuration) and the address of the Kafka server is passed as an environment variable. The Master machine now logs into the Slave machine, specifies the number of cores and memory to be given to a node and triggers a Docker container.

Docker specific design choices:

1. **Networking**: In order to circumvent the local network created by the Docker daemon, the Docker container, in which the node runs, shares the networking stack with the host Slave machine.

2. **Storage**: For testing the model accuracy, we need access to raw data files which contain the sensor data. If we include the data files in building the image, the image size will blow up to tens of gigabytes. As a host Slave can support multiple Docker containers and to reduce the Docker image size, the sensor data is kept on the host's file system and the directory is bind to an empty directory of the Docker image.

3. **File reading**: In order to avoid reading the whole data file (few Gigabytes) into the memory of a running container (hundreds of Megabytes), an optimization has been done to sample data points from the file by seeking to lines arbitrarily.

6 Running the Simulator

The Simulator can be run in two modes: Docker mode and non-Docker mode.

6.1 Configuration setup

Every experiment has a corresponding configuration file in the form of yaml.

```yaml
delays:
  - src_id: 1
    dest_id: 2
    delay: 0

default_delay: 0
default_mini_batch_size: 100
default_window_interval: 5
default_window_limit: 2
default_epochs_per_window: 1
default_kafka_server: "10.152.50.10:9092"

default_cpus: 1
default_memory: "125M"

default_docker_image: "aniketshirke/distributedlearning:simulator"
default_host_test_directory: "~/Simulator/TreeNN/data"
default_test_directory: "/TreeNN/data/"

machine:
-   ip: 10.129.2.26
    username: "synerg"
    password: "synerg"

nodes:
-   id: 1
    port: 8000
    machine: 0
-   id: 2
    port: 8006
    machine: 0
    parent_id: 1
    mini_batch_size: 10
    memory: "125M"
    sensors: [1,2]

```

Common fields for configuration:

- `delays`: Link delays between nodes specified pairwise
- `machine`: IP and credentials of the machines across which the simulation is run
- `default_delay`: Default link delay between nodes
- `default_mini_batch_size`: Default mini batch size for training (Field depends on the algorithm used)
- `default_window_interval`: Default time interval for which data is collected by worker node

- `default_window_limit`: Default number of windows processed by worker during the experiment
- `default_epochs_per_window`: Default number of epochs to run on training data collected in a window
- `default_kafka_server`: Default address of Kafka server for worker node
- `default_cpus`: Default cores to be allocated to each node (Docker-specific)
- `default_memory`: Default memory to be allocated to each node (Docker-specific)
- `default_docker_image`: Default image to be used to instantiate the container (Docker-specific)
- `default_host_test_directory`: Default directory on the host Slave which contains the test data, required for binding (Docker-specific)

Fields for configuring nodes:

- `id`: Node ID
- `machine`: ID of the Slave machine
- `port`: Port on which RPC server is run
- `test_directory`: Directory which contains test data. If run on a Docker container, it is a placeholder for binding with the host test directory. Else it is an actual directory on the host
- `sensors`: Array of sensors to which Worker will subscribe
- `window_interval` (Optional)
- `window_limit` (Optional)
- `epochs_per_window` (Optional)
- `kafka_server` (Optional)
- `cpus` (Optional, Docker-specific)
- `memory` (Optional, Docker-specific)
- `docker_image` (Optional, Docker-specific)
- `host_test_directory` (Optional, Docker-specific)

6.2 Commands

The following are the steps to be taken to run the simulator:

1. Create a configuration file adhering to the specification given above.
2. Start Kafka service on a server
3. Run `master.py` on the Master machine. The experiment can be run on two modes:
    - Docker mode: Set the `--docker` flag to run. The configuration file has to be set up accordingly.
    - Server mode: In this mode, there will be no restriction on the resources provided in simulating Edge nodes.
4. Run the script which simulates the sensor.

7 Experiments

7.1 Log format

Every Slave reports back to the Master by providing logs in JSON format. The structure of the log is as follows:

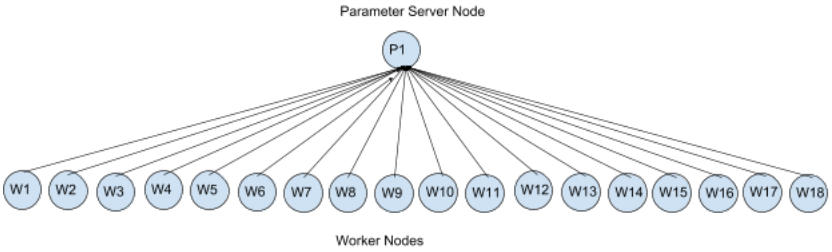
- `NODE_ID`: ID of the node which reported the log
- `TYPE`: There are three types of logs:
    - `CONNECTION`: Log indicates the communication of the node with any other entity: parent, Master or Kafka server
    - `STATISTIC`: Log contains statistics about the simulation. Refer here
    - `DONE`: Special 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

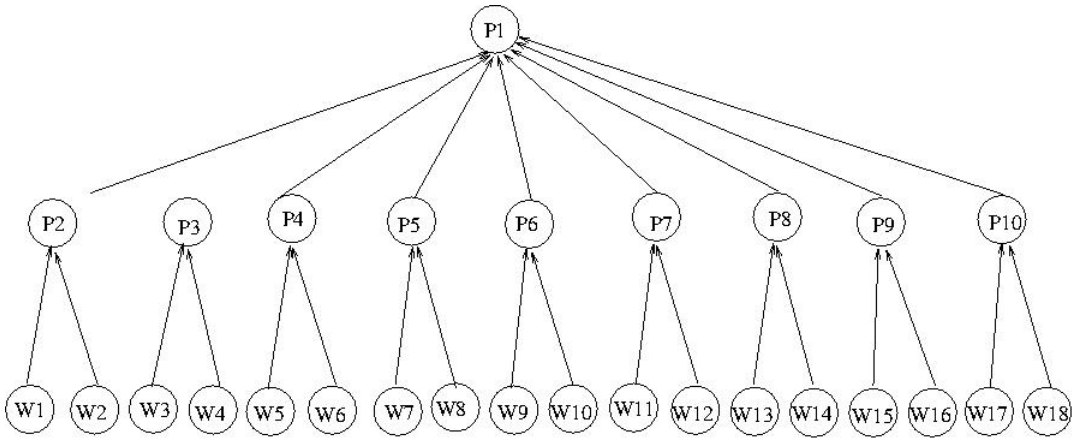
7.2 Distributed Computing Hierarchies-[5]

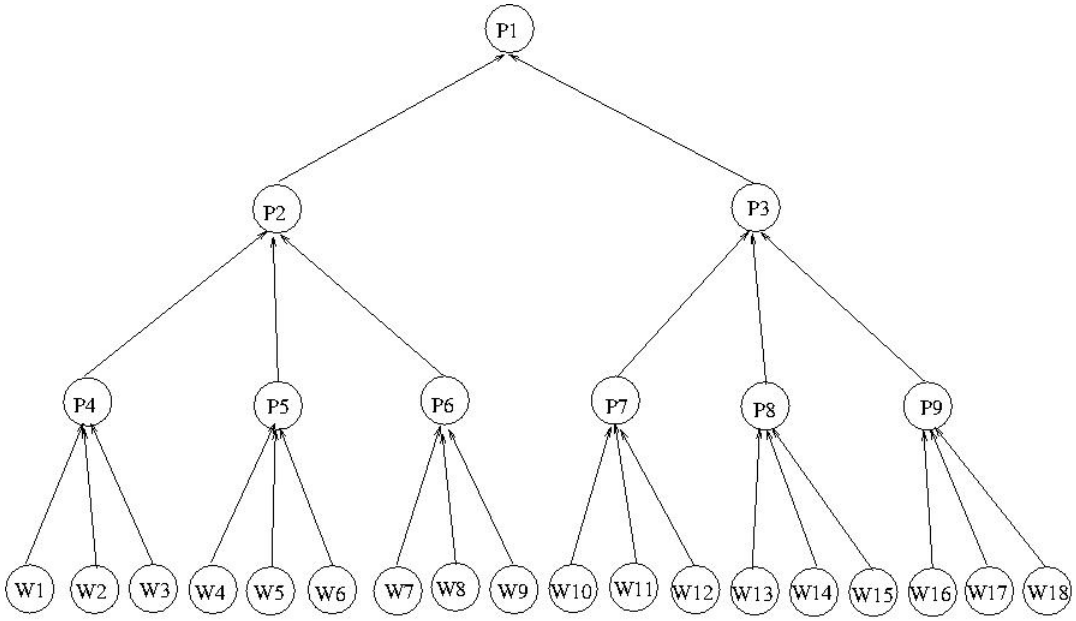
These are the ongoing experiments which are going to be conducted with the help of the Simulator.

Here we assume that our worker nodes have same resources and all parameter servers have same resources. Our model is learnt continuously.

Experiment 1: Varying Worker nodes' configuration
For each worker node configuration, find the processing time per data point and accordingly set the window time duration and window size for different worker node configuration settings. Vary the worker node configuration from 1 to 4 core CPUs and report the variation in all the metrics for a given configuration setup.







Experiment 2: Varying number of parameter nodes for a fixed level of hierarchy
Show the variation in all the metrics for Configuration setup 2 and compare with the Configuration setup 1. For accuracy cost, we reduce the number of spatial neighborhood queries to 2 or 3 to show the comparison in one plot.

Experiment 3: Varying the link delays
Vary the link delays for all configuration setups and compare the accuracy lag across every setup

Experiment 4: Model Exchange Policy
For a given configuration and fixed data rate, we can set some limits for sending data to its parent. For example: if the difference in test accuracy of the current model and previous model is within the threshold then there is no need to send the data to its parent. Same test can be done when a child node asks for the model from its parent. If the difference in accuracy of the current and the previous model sent to the child is within the threshold then parent will not transfer the model. Another threshold could be time threshold.

7.3 Metrics-[5]

We measure the following two different types of cost of a given distributed computing hierarchy using the Simulator:

7.3.1 Model Performance

1. **Accuracy**: Each worker node learns the model using the local data available to it from sensors. It is expected that the worker node can answer prediction queries of own cell with high accuracy instead of queries of distant cells because it will take some time to transfer the learning between two distant worker nodes. To measure this time, we perform the prediction queries with respect to space and time, and measure the error (RMSE) using the truth value. A prediction query is AQI(time, location)

2. **Latency**: Latency is the maximum time lag among all worker nodes where time lag of a worker node is the time difference between model update by last worker node using the local data at time window j and the updated model received from the parent node which has been updated by all other worker nodes using their local data of time window j.

Out of n Worker nodes, consider the i^{th} worker node and let j be the time window varying from 1 to T.
t_j^i : timestamp when j^th time window is processed at worker node i
tr_j^i: smallest timestamp when worker node i receives the model from its parent which is update using time window j or greater.
For each worker node i, we define latency at window j as follows: $latency_j^i = min_{k\in[1,n]}(tr_j^i - t_j^k)$
For each worker node i, we define latency as follows: $latency^i = max_{j\in[1,T]}(latency_j^i)$
Latency of the given configuration: $latency = max_{i\in[1,n]}(latency^i)$

7.3.2 Computational Performance

1. **CPU cost**: Total CPU time across all nodes to train the model over the complete data.

2. **Memory Cost**: Total memory usage across all nodes

3. **Network Cost**: Total network cost across all links

8 Future Work

8.1 Front End

The backend portion of the Simulator is implemented in Python3 currently. To enhance ease of use, a desirable feature is a user interface on which the progress of an ongoing simulator is displayed in the form of animated graphs. Some work was done to develop a basic GUI application as a part of the project using Tkinter in Python. The idea was dropped as Tkinter is quite inflexible in terms of providing ease of development and many modern user interfaces are web applications, e.g. Neo4j. We intend to develop a Python based Flask application to build the Front End for the simulator where the user can input the experiment configuration and view the results of the ongoing experiment.

8.2 Optimizations to the existing Simulator

Further research on all the ways of model and data sharing for training the neural network is required. Experiments need to be conducted to figure out the optimal sharing strategy minimizing the communication costs and maximizing the accuracy of the system.

8.3 Extend to other domains of learning

The simulation environment has been used for learning the AQI model, which is a supervised learning problem. The simulator can be extended to other supervised learning problems and other applications need to be built to show the efficacy. The simulator needs to be integrated with other unsupervised learning problems.

8.4 Data Sampling and Distribution Strategy

If the data generation rate by the sensors is very high, then the Edge devices need to be computationally more powerful, else they might be a bottleneck. [7] and [8] have proposed online sampling strategies to incoming streaming data. Such strategies can be explored with the help of the simulator to sample and distribute data streams efficiently amongst Edge centers.

9 Conclusion

Advancements in Edge Computing and IoT analytics is happening at a rapid pace. We apply the Simulator on the Air Quality Index prediction problem. Steps have been taken to ensure that the Simulator simulates real scenarios. Extensive experimentation with the use of the Simulator is in progress. The Simulator promises to be a tool which can be used to find optimal hierarchical structure of Edge centers, given the budget constraints, for any real life IoT application.

10 Acknowledgement

I would like to thank Govind Lahoti who initiated this work and Alka Bhushan for constantly helping me out in different phases of the project, including data generation for different sensors. I wish to express my sincere gratitude to Prof. Umesh Bellur for his constant guidance and support.

References

[1] Mohammadi, Mehdi, Ala Al-Fuqaha, Sameh Sorour, and Mohsen Guizani. "Deep Learning for IoT Big Data and Streaming Analytics: A Survey." IEEE Communications Surveys Tutorials (2018).

[2] Teerapittayanon, Surat, Bradley McDanel, and H. T. Kung. "Distributed deep neural networks over the cloud, the edge and end devices." In Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on, pp. 328-339. IEEE, 2017.

[3] Wang, Shiqiang, Tiffany Tuor, Theodoros Salonidis, Kin K. Leung, Christian Makaya, Ting He, and Kevin Chan. "When edge meets learning: Adaptive control for resource-constrained distributed machine learning." arXiv preprint arXiv:1804.05271 (2018).

[4] Govind Lahoti, "Scheduling Lambda Functions on the IoT Edge", BTech Project Report

[5] The sections have been borrowed from Alka Bhushan's writeup: `https://goo.gl/cNePVA`

[6] Article:`https://techcrunch.com/2018/02/07/intels-latest-chip-is-designed-for-computing-at-the-edge/`

[7] Wen, Zhenyu, Pramod Bhatotia, Ruichuan Chen, and Myungjin Lee. "ApproxIoT: Approximate Analytics for Edge Computing." In 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), pp. 411-421. IEEE, 2018.

[8] Quoc, Do Le, Ruichuan Chen, Pramod Bhatotia, Christof Fetzer, Volker Hilt, and Thorsten Strufe. "StreamApprox: approximate computing for stream analytics." In Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, pp. 185-197. ACM, 2017.