

Middle East Technical University
Department of Computer Engineering
Wireless Systems, Networks and Cybersecurity (WINS) Laboratory



Distributed Learning

CENG797 Ad Hoc Networks
2020-2021 Fall
Term Project Report

Prepared by
Berker Acir
Student ID: 2098697
berker.acir@metu.edu.tr
Computer Engineering
6 January 2021

Abstract

In this term project report, distributed learning algorithms' implementations and testings on ad hoc network simulation is described. The studied distributed algorithms are for constructing minimum spanning tree and training a Machine Learning model collaboratively in the network nodes. Anytime Distributed MST algorithm is implemented for constructing minimum spanning tree and Reliable Parameter Server algorithm is implemented for training a Sequential Deep Neural Network Model within the ad hoc nodes. Minimum spanning tree algorithm works well in a fixed network regardless of node count. Reliable Parameter Server algorithm is observed to have positive effects on the learning process. Combining the two implementations seems to be promising as it can be used for reducing the communication costs and bandwidth allocation.

Table of Contents

Abstract	ii
List of Figures	iv
List of Tables	v
1 Introduction	1
2 Background and Related Work	1
2.1 Background	1
2.1.1 Machine Learning	1
2.1.2 Deep Learning	2
2.2 Related Work	2
2.2.1 Anytime Distributed Minimum Spanning Tree Algorithm	2
2.2.2 Reliable Parameter Server Algorithm	4
3 Project Implementation	7
3.1 Components	7
3.1.1 Node	7
3.1.2 LinkComponent	7
3.1.3 MSTComponent	7
3.1.4 RPSCComponent	10
3.2 Channels	10
3.2.1 NodeChannel	10
4 Results and Discussion	11
4.1 Methodologies	11
4.1.1 NodeChannel	11
4.1.2 MSTComponent	12
4.1.3 RPSCComponent	12
4.2 Results	13
4.2.1 Multiple Model Training Out of the Network Simulation	13
4.2.2 Multiple Model Training in the Network Simulation	13
4.3 Discussion	14
5 Conclusion	14

List of Figures

Figure 1	Deep Neural Network's layered architecture	2
Figure 2	Components View	7
Figure 3	Components' Event and Message Types	7
Figure 4	(a) Initial Network Topology (b) Constructed Minimum Spanning Tree (c, d, e, f, g, h) Running of Anytime Distributed MST Algorithm started from node with id 2 where blue nodes are not in the tree yet, red and green toned nodes are in the tree. Red color shows that the node is not activated yet whereas green color shows that the node is activated. Darker green color means that the node is the last activated node.	9
Figure 5	Help command shows available commands.	11
Figure 6	Complex Sequential Model Summary	12
Figure 7	Simple Sequential Model Summary	12

List of Tables

Table 1	Multiple Model Training Out of the Network Simulation	13
Table 2	Multiple Model Training in the Network Simulation	14

1 Introduction

With the rapid development of new technologies, data collection has grown unprecedentedly [1]. Together with it, the need for processing and making sense out of the collected data has increased. As computing and storing capabilities of end devices such as IoT devices and smartphones are soaring, data processing paradigm is shifting towards the end devices or network edges from cloud servers [2] in order to decrease the burden on back-bone network [3].

In this project, I implemented one of Machine Learning algorithm on an ad hoc network simulation as Machine Learning algorithms are proven for making sense of the big data and crucial part of distributed learning. Running Machine Learning algorithms on nodes instead of in servers can be beneficial in terms of reducing the network traffic. Instead of sharing whole collected data to server, the nodes will run the algorithm over data and send learned model to the server. However, implementing machine learning algorithms in distributed manner are not always reasonable due to the convergence problems of model.

I also implemented Minimum Spanning Tree algorithm in my project as learnt models or their weights can be distributed to other nodes using the tree with the thought of reducing communication between the nodes. Therefore, Minimum Spanning Tree must be constructed before the beginning of learning process.

My project has two main components: one for constructing Minimum Spanning Tree and starting the learning process, and the other one for continuing the learning while sharing the learnt parameters with other nodes in the network. As some of the Machine Learning algorithms are too complex for my hardware capabilities, I had to select a simpler Machine Learning task as the simulation will run different numbers of nodes in parallel manner on single CPU and GPU.

The rest of this report is organized as follow. I first give background information followed by related works which I implemented in the network simulation in Section 2. Then, I explain implementation details of my project with detailed information on used components and channels in Section 3. In Section 4, I further explain my methodologies, share my results and opinions. At last, I conclude the project in Section 5.

2 Background and Related Work

2.1 Background

2.1.1 Machine Learning

Machine Learning, subset of Artificial Intelligence, is the self learning process of computer through examples [4]. Machine learning algorithms construct a model based on training data in order to make predictions, decisions or inferences without being explicitly programmed to do so. Machine learning algorithms are usually closely related to computational statistics. Complex tasks can be challenging to be handled by traditional task-specific algorithms. As it turned out that it is much more effective to help the machine to develop its own algorithm instead of tailor-one by hand [5].

There exist different types of Machine Learning algorithms due to differences in their approach, the type of data they input and output, and the type of task or problem that they are intended to solve. Some of the Machine Learning types:

- **Supervised Learning:** The model has example inputs and the related outputs at present with the goal of learning a general rule for transforming input variables to desired output.
- **Unsupervised Learning:** The model has example inputs without any label outputs which leaves the model on its own to discover hidden patterns in its input.
- **Reinforcement Learning:** The model interacts with dynamic environment which provides feedback to the learning process with the goal of maximizing positive feedback [6].

2.1.2 Deep Learning

Conventional Machine Learning algorithms rely on hand-engineered feature extractors to process raw data [7]. Feature selection has to be customized and re-initiated for each different problem. Otherwise, Deep Neural Networks can automatically discover and learn features from raw data [8]. Therefore, Deep Learning often outperforms conventional ML algorithms especially when the data is abundant.

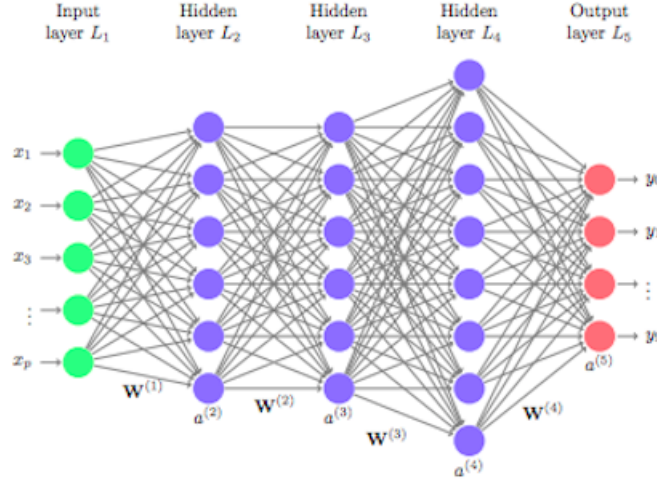


Figure 1 . Deep Neural Network's layered architecture

Deep Learning lies within the domain of the brain-inspired computing paradigm of which the neural network is an important part. As illustrated in figure 1 , the neural network comprises three layers: (i) input layer, (ii) hidden layer, and (iii) output layer. A typical Deep Neural Network comprises multiple hidden layers that map an input to an output. The objective of training a Deep Neural Network is to optimize the weights of the network such that the loss function is minimum.

2.2 Related Work

The two algorithms that I implemented in my term project:

2.2.1 Anytime Distributed Minimum Spanning Tree Algorithm

Authors in [9], states the problem and solution as the following: Let $G = (V, E, W)$ be a weighted, undirected connected graph and each node $v \in V$ is equipped with (i) a data

structure representing the same, initial spanning tree $S_0 = (V, E_{S_0}, W_{S_0})$ of G , (ii) a list of its immediate neighbors and the associated edge weight in G . Then, a distributed recursive minimum spanning tree policy is initiated at an arbitrary "active" node $a_o \in V$, whose next, $k + 1$ step, follows from step $k \in \mathbb{N}$ with the following properties:

- *Local Optimization*: at each step, k , node a_k computes a locally optimal restructuring of S_k towards another spanning tree S_{k+1} that has a lower edge weight sum;
- *Broadcast*: at each step, k , node a_k broadcasts to all nodes the local edge insertions and deletions that transform S_k into S_{k+1} and then activates an adjacent node, $a_{k+1} \in V$.

Clearly with such policy (algorithm 1) at each step, k , there is only one active node, $a_k \in V$, and it shows that the recursion halts at a MST exactly when every node has been activated at least once with the $O(|V||E|)$ worst case computational and communications costs.

Authors also introduce some notation before the algorithms. A spanning tree $S = (V, E_S, W_S)$ defines a unique path between any pair of nodes. Let $P_S(u, v)$ denote the path joining node u and v in a spanning tree S , and $d_S(u, v)$ denote the hop distance between u and v defined as the maximum hop length along $P_S(u, v)$,

$$d_s(u, v) := \max_{(i,j) \in P_S(u,v)} W_s(i, j), \quad (1)$$

and let $e_s(u, v)$ be an edge in $P_S(u, v)$ with the maximum hop length,

$$e_s(u, v) := \operatorname{argmax}_{(i,j) \in P_S(u,v)} W_s(i, j), \quad (2)$$

Note that the hop distance of d_s of an active node to all other nodes is needed, which can be computed in linear time, $O(|V|)$. Also, it is convenient to have $N_G(u)$ be the ordered set of adjacent nodes of node u in $G = (V, E, W)$ in ascending order according to the associated edge cost, i.e.

$$N_G(u) := (v \in V | (u, v) \in E) = (v_1, v_2, \dots, v_{\deg_G(u)}), \quad (3)$$

$$W(u, v_i) \leq W(u, v_j) \iff i \leq j, \quad (4)$$

where $\deg_G(u)$ denotes the degree of node u in G . Finally, let $G|_F := (V, F, W_F)$ denote the sub-graph of graph $G = (V, E, W)$ with edge set $F \subseteq E$ and edge weights $W_F(u, v) = W(u, v)$ for all $(u, v) \in F$.

Algorithm 1 Locally Optimal Spanning Tree Restructuring Policy (LORE)

Require: $S = (V, E_S, W_S)$ be a spanning tree of a connected graph $G = (V, E, W)$, and $u \in V$ be the active node. Then, the set of locally optimal edge deletions D and insertions I of S are found as follows:

- 1: Begin with $D = \emptyset$, $I = \emptyset$, and $S_1 = S$.
 - 2: **for** every $k \in [1, \deg_G(u)]$ and $v = N_G(u)_k$ **do**
 - 3: **if** $W(u, v) < d_{S_k}(u, v)$ **then**
 - 4: $D = D \cup \{e_{S_k}(u, v)\}$
 - 5: $I = I \cup \{(u, v)\}$
 - 6: $S_{k+1} = G|_{(E_S \cup I) \setminus D}$
 - 7: **end if**
 - 8: **end for**
-

Algorithm 2 Anytime Distributed MST Algorithm

Require: $S = (V, E_S, W_S)$ be a spanning tree of a connected graph $G = (V, E, W)$ shared by all nodes in V , and $a_0 \in V$ be the initial active node that is assumed to receive a token from $a_{-1} \in N_G(a_0)$.

- 1: Begin with $S_0 = S$.
 - 2: **for** every $k \in \mathbb{N}$ until the activation counter reaches $|V|$ **do**
 - 3: Find locally optimal restructuring of S_k at node a_k , $(D_k, I_k) = \text{LORE}(S_k, a_k)$ (Algorithm 1)
 - 4: Update $S_{k+1} = G|_{(E_{S_k} \cup I_k) \setminus D_k}$
 - 5: Broadcast the found updates (D_k, I_k) , with associated edge weights, through S_{k+1}
 - 6: Wait until every node receives the updates
 - 7: Pass the token to the next node, $a_{k+1} = \text{TC}_{S_{k+1}}(a_{k-1}, a_k)$
 - 8: **end for**
-

2.2.2 Reliable Parameter Server Algorithm

Definitions and notations given by the authors in [10]:

- $\nabla f(\cdot)$ denotes the gradient of the function f .
- $\lambda_i(\cdot)$ denotes the i th largest eigenvalue of a matrix.
- $\mathbf{1}_n = [1, 1, \dots, 1]^T \in \mathbb{R}^n$ denotes the full-one vector.
- $A_n := \frac{\mathbf{1}_n \mathbf{1}_n^T}{n}$ denotes the all $\frac{1}{n}$'s n by n matrix.
- $\|\cdot\|$ denotes the ℓ_2 norm for vectors.
- $\|\cdot\|_F$ denotes the Frobenius norm of matrices.

In the RPS algorithm, each node maintains an individual local model. Authors used $x_t^{(i)}$ for denoting the local model on node i at time step t . At each iteration t , each node first performs a regular Stochastic Gradient Descent (SGD) step

$$v_t^{(i)} \leftarrow x_t^{(i)} - \gamma \nabla F_i(x_t^{(i)}; \xi_t^{(i)}); \quad (5)$$

where γ is the learning rate and $\xi_t^{(i)}$ are the data samples of node i at iteration t .

Main objective is reliably averaging the vector $v_t^{(i)}$ among all nodes via the RPS procedure. In brief, the *Reduce-Scatter* step performs communication-efficient model averaging, and the *AllGather* step performs communication-efficient model sharing.

- **The Reduce-Scatter (RS) step:** Each node i divides $v_t^{(i)}$ into n equally-sized blocks.

$$v_t^{(i)} = \left((v_t^{(i,1)})^T, (v_t^{(i,2)})^T, \dots, (v_t^{(i,n)})^T \right)^T. \quad (6)$$

The vector $v_t^{(i)}$ is divided in order to reduce the communication cost and parallelize model averaging.

After the division, each node sends its blocks to corresponding node. Upon receiving blocks, each node averages all the blocks it receives. Some packets might be dropped; in this case, node i averages all those blocks using

$$\tilde{v}_t^{(i)} = \frac{1}{|\mathcal{N}_t^{(i)}|} \sum_{j \in \mathcal{N}_t^{(i)}} v_t^{(i,j)}, \quad (7)$$

where $\mathcal{N}_t^{(i)}$ is the set of packages node i receives including the node i 's own packages.

- **The AllGather (AG) step:** After computing $\tilde{v}_t^{(i)}$, each node i attempts to broadcast $\tilde{v}_t^{(i)}$ to all other nodes, using the averaged blocks to recover the averaged original vector $v_t^{(i)}$ by concatenation:

$$x_{t+1}^{(i)} = \left((\tilde{v}_t^{(i,1)})^T, (\tilde{v}_t^{(i,2)})^T, \dots, (\tilde{v}_t^{(i,n)})^T \right)^T. \quad (8)$$

It is entirely possible that some nodes in the network may not be able to receive some of the averaged blocks. In this case, they just use the original block. Formally,

$$x_{t+1}^{(i)} = \left((x_{t+1}^{(i,1)})^T, (x_{t+1}^{(i,2)})^T, \dots, (x_{t+1}^{(i,n)})^T \right)^T, \quad (9)$$

where

$$x_{t+1}^{(i,j)} = \begin{cases} \tilde{v}_t^{(j)} & j \in \tilde{\mathcal{N}}_t^{(i)} \\ v_t^{(i,j)} & j \notin \tilde{\mathcal{N}}_t^{(i)} \end{cases} \quad (10)$$

We can see that each node just replaces the corresponding blocks of $v_t^{(i)}$ using received averaged blocks.

Algorithm 3 Reliable Parameter Server

Require: Initialize all $x_1^{(i)}$, $\forall i \in [n]$ with the same value, learning rate γ , and number of total iterations T .

- 1: **for** $t = 1, 2, \dots, T$ **do**
- 2: Randomly sample $\xi_t^{(i)}$ from local data of the i th node, $\forall i \in [n]$
- 3: Compute a local stochastic gradient based on $\xi_t^{(i)}$ and current optimization variable $x_t^{(i)} : \nabla F_i(x_t^{(i)}; \xi_t^{(i)})$, $\forall i \in [n]$
- 4: Compute the intermediate model $v_t^{(i)}$ according to

$$v_t^{(i)} \leftarrow x_t^{(i)} - \gamma \nabla F_i(x_t^{(i)}; \xi_t^{(i)}),$$

and divide $v_t^{(i)}$ into n blocks

$$\left((v_t^{(i,1)})^T, (v_t^{(i,2)})^T, \dots, (v_t^{(i,n)})^T \right)^T.$$

- 5: For any $i \in [n]$, randomly choose one node $b_t^{(i)}$ without replacement. Then, every node attempts to send their i th block of their intermediate model to node $b_t^{(i)}$. Then, each node averages all received blocks using

$$\tilde{v}_t^{(i)} = \frac{1}{|\mathcal{N}_t^{(i)}|} \sum_{j \in \mathcal{N}_t^{(i)}} v_t^{(i,j)}.$$

- 6: Node $b_t^{(i)}$ broadcast $\tilde{v}_t^{(i)}$ to all nodes and packets maybe dropped, $\forall i \in [n]$.
- 7: $x_{t+1}^{(i)} = \left((x_{t+1}^{(i,1)})^T, (x_{t+1}^{(i,2)})^T, \dots, (x_{t+1}^{(i,n)})^T \right)^T$, where

$$x_{t+1}^{(i,j)} = \begin{cases} \tilde{v}_t^{(j)} & j \in \tilde{\mathcal{N}}_t^{(i)} \\ v_t^{(i,j)} & j \notin \tilde{\mathcal{N}}_t^{(i)} \end{cases}$$

for all $i \in [n]$.

- 8: **end for**
-

3 Project Implementation

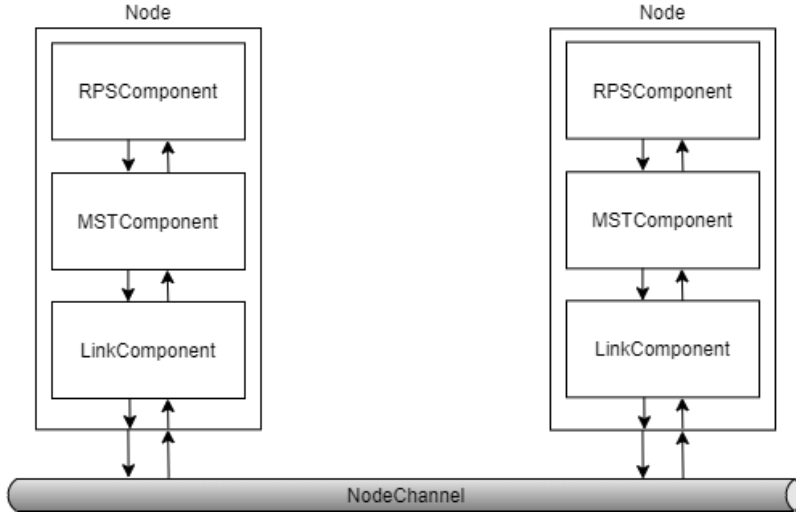


Figure 2 . Components View

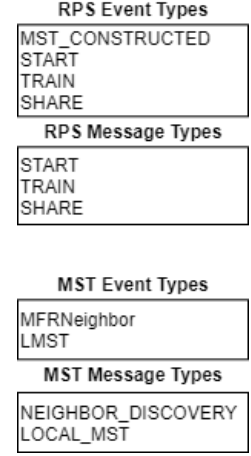


Figure 3 . Components' Event and Message Types

I designed my components as illustrated in figure 2 where *Nodes* are connected to each other through *NodeChannels* which work in peer-to-peer fashion. *Nodes* have *RPS*, *MST* and *Link* components internally, and their hierarchy is as shown in the figure 2. *RPS* and *MST* components use their own event and messaging types which are shown in figure 3.

3.1 Components

3.1.1 Node

It contains internal components and it's connected to LinkComponent and also NodeChannel. It configures the connections of components within the node itself. It also connects the internal components to other node's components via NodeChannel.

3.1.2 LinkComponent

It is in the bottom layer in node's internal components and it's connected to upper layer MSTComponent and the Node itself.

3.1.3 MSTComponent

It's connected to upper layer RPSComponent and lower layer LinkComponent. It's responsible for discovering neighbors, computing local minimum spanning tree, broadcasting the updated its local minimum spanning tree to other nodes. MSTComponent also responsible for selecting activation of next node. MSTComponent implements **Anytime Distributed MST Algorithms** 1 and 2. After minimum spanning tree constructed, MSTComponent notices the RPSComponent of the same node which results in the starting of the learning process, i.e. RPS Algorithm.

MSTComponent have the following event types:

- **MFRNeighbor (Message From Neighbor):** This event occurs when the component receives neighbor discovery messages. This event is used for discovering neighbors and the discovered neighbors are used in local minimum spanning tree construction.
- **LMST (Local Minimum Spanning Tree):** This event fires when the component receives local minimum spanning tree messages from the other nodes. Depending on the message and the node's activation status, the component either updates its own local tree and pass the incoming message towards the other direction in the tree or calculates its own local tree (which means node is activated), broadcasts its local minimum spanning tree to other nodes in the tree and selects a new node for activation.

MSTComponent have the following message types:

- **NEIGHBOR_DISCOVERY (Neighbor Discovery):** This message type indicates that related message contains neighbor information such as its node id and the cost (i.e. weight) of edge between the nodes.
- **LOCAL_MST (Local Minimum Spanning Tree Update):** This message type is used for local minimum spanning tree updates. It is used for forwarding the received minimum spanning tree updates, broadcasting calculation of local minimum spanning tree, and also broadcasting of next node that will be activated.

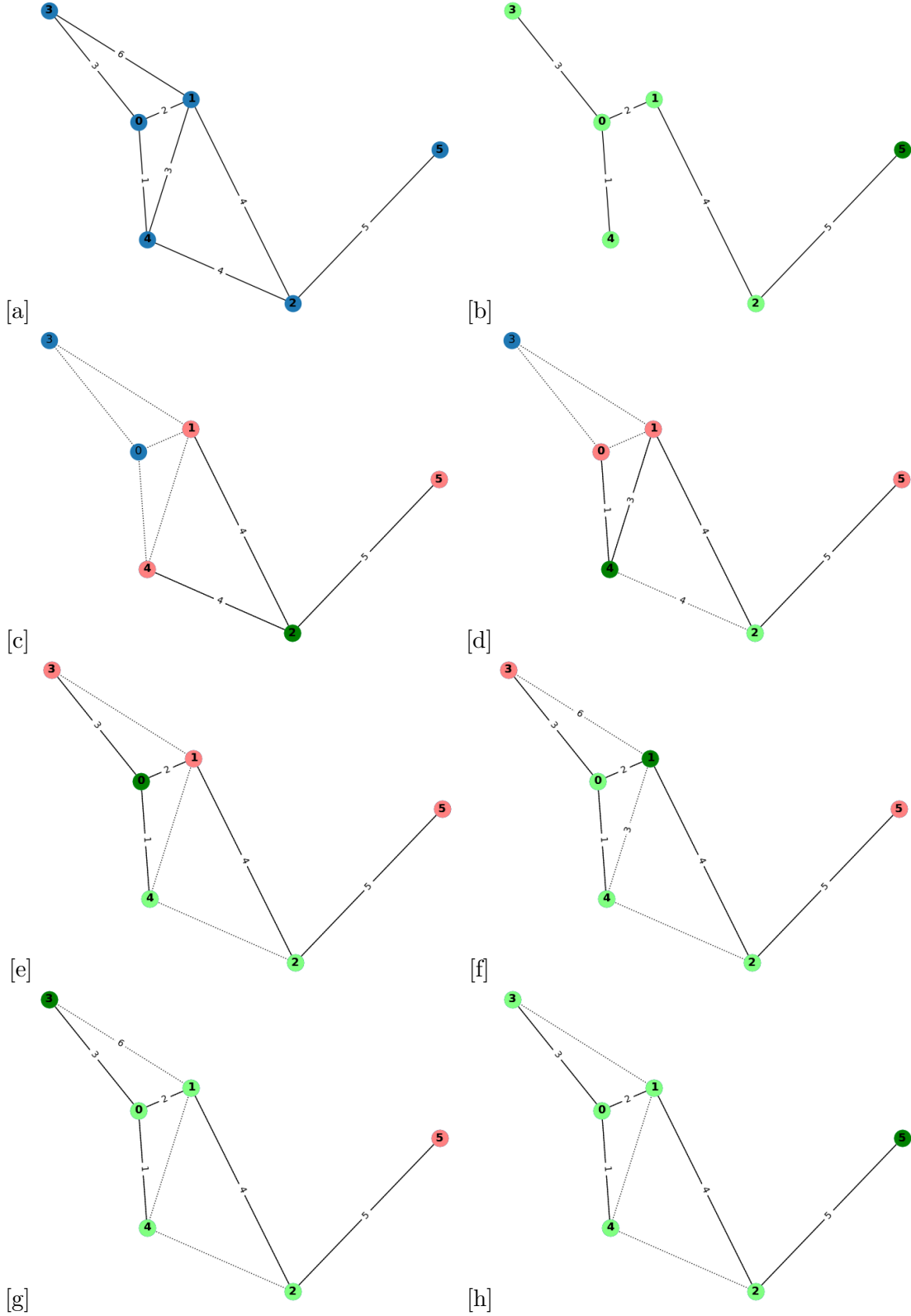


Figure 4 . (a) Initial Network Topology (b) Constructed Minimum Spanning Tree (c, d, e, f, g, h) Running of Anytime Distributed MST Algorithm started from node with id 2 where blue nodes are not in the tree yet, red and green toned nodes are in the tree. Red color shows that the node is not activated yet whereas green color shows that the node is activated. Darker green color means that the node is the last activated node.

After the minimum spanning tree is constructed, the MSTComponent is responsible for noticing the RPSComponent with its local minimum spanning tree update message which results in beginning of distributed learning process.

3.1.4 RPSComponent

It is in the top layer in node's internal components and it's only connected to MSTComponent. RPSComponent is responsible for running **Reliable Parameter Server** application included with communication with other node's RPSComponents.

When a RPSComponent is noticed with local minimum spanning tree update message by lower layer MSTComponent, it initiates learning process by sending start messages to other nodes' RPSComponent using local minimum spanning tree. After that, learning process is started in an distributed fashion. Each component shares its own learnt parameters periodically until the learning process is halt or converges.

RPSComponent have the following event types:

- **MST_CONSTRUCTED (Minimum Spanning Tree Constructed):** This event occurs when the component receives local minimum spanning tree update message. Then, the node's start event is fired.
- **START (Start):** This event happens when the component receives RPS start message. It starts the learning process with broadcasting of RPS start message to other nodes in the local minimum spanning tree, and also starts the training process.
- **TRAIN (Train):** This event is used for starting and continuing the training phase.
- **SHARE (Share):** This event is used for sharing of learnt parameters (i.e. weights) to other nodes in the tree.

RPSComponent have the following message types:

- **START (Start):** This message type contains received constructed minimum spanning tree information and it is used for starting learning process and broadcasting beginning of the training process. Start messages are broadcasted through the tree; in other words, when a node receives start message from one of its spanning tree neighbor, it forwards the message to other spanning tree neighbors except the source node.
- **TRAIN (Train):** This message type does not used for message carrying; yet, it is used for train event within the RPSComponent. When the component receives train message, it fires train event. Therefore, this message is used for starting or continuing the learning process.
- **SHARE (Share):** This message type contains shared weights with the source and forwarded node's information. It is crucial for RPS algorithm as it contains the shared weights.

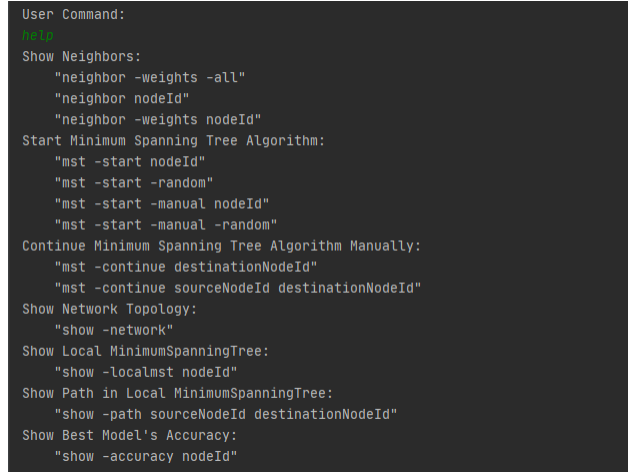
3.2 Channels

3.2.1 NodeChannel

I used only one channel which is named *NodeChannel*. It works in peer-to-peer manner as I thought the learning process will be implemented through minimum spanning tree in order to share parameters with the minimum cost among all nodes.

4 Results and Discussion

4.1 Methodologies



```
User Command:
help
Show Neighbors:
"neighbor -weights -all"
"neighbor nodeId"
"neighbor -weights nodeId"
Start Minimum Spanning Tree Algorithm:
"mst -start nodeId"
"mst -start -random"
"mst -start -manual nodeId"
"mst -start -manual -random"
Continue Minimum Spanning Tree Algorithm Manually:
"mst -continue destinationNodeId"
"mst -continue sourceNodeId destinationNodeId"
Show Network Topology:
"show -network"
Show Local MinimumSpanningTree:
"show -localmst nodeId"
Show Path in Local MinimumSpanningTree:
"show -path sourceNodeId destinationNodeId"
Show Best Model's Accuracy:
"show -accuracy nodeId"
```

Figure 5 . Help command shows available commands.

I found it hard to debug my codes because the given Event Driven Component Model works with multi-threads. Therefore, I thought of creating command shell for the term project. Example output of *help* command can be seen in the figure 5

- **neighbor** Command: It is used for obtaining neighbor information of nodes. It can take *-weights* and *-all* arguments along with *nodeId*. Weight argument is used for showing the weights and all argument is used for showing all neighbor information instead of showing just one *nodeId*'s neighbors.
- **mst** Command: It is used for starting and continuing minimum spanning tree algorithm. It can take *-start*, *-random*, *-manual* and *-continue* arguments along with *nodeId*, *sourceNodeId* and *destinationNodeId*. Start argument is used for starting the algorithm and the algorithm can be continued manually if the algorithm is started with manual argument. If not, algorithm will run automatically and after construction of the tree, it will continue to starting the learning phase. Starting node can be assigned randomly, if random argument is presented; otherwise, *nodeId* must be given. If the algorithm started manually, it can be continued with continue argument which takes *destinationNodeId* only or *sourceNodeId* together with *destinationNodeId*. Minimum spanning tree algorithm will draw the latest local tree.
- **show** Command: It is used for reaching the local variables when the program is running. It can take *-network*, *-localmst*, *-path* or *-accuracy* arguments along with *nodeId*, or *sourceNodeId* together with *destinationNodeId*.

4.1.1 NodeChannel

The NodeChannel changes (i.e., updates) only neighbor discovery messages. I assigned random weight information to channels and the channels share their weight (cost) through updating neighbor discovery messages. By this way, neighbors learn its channels weights with discovering their neighbors.

I also added intentional sleep to channel when the message type is local minimum spanning tree update. That was my solution to drawing minimum spanning trees as soon as the new node is activated. Otherwise, it was throwing errors related with drawing.

4.1.2 MSTComponent

Anytime Distributed MST algorithm is designed for fixed networks, and the authors assume that all nodes have the information of general topology. However, I did implement similar algorithm with sharing only locally inserted (*I*) and deleted (*D*) edge and activated node information where the nodes does not know the topology. First, the nodes discover their surroundings (i.e., neighbor nodes), then minimum spanning algorithm starts from one of the nodes. In the paper [9], they used the Euler tour for selecting next node to be activated; however, in my implementation, the last activated node will select the next closest inactive node in terms of weight and hop counts. I also tried to implement ACK mechanism, but I faced deadlock problems which I couldn't solve in time. It resides in my git branch.

4.1.3 RPSComponent

Model: "sequential"		
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 30, 30, 32)	896
conv2d_1 (Conv2D)	(None, 28, 28, 64)	18496
conv2d_2 (Conv2D)	(None, 26, 26, 128)	73856
flatten (Flatten)	(None, 86528)	0
dense (Dense)	(None, 128)	11075712
dense_1 (Dense)	(None, 10)	1290
Total params: 11,170,250		
Trainable params: 11,170,250		
Non-trainable params: 0		

Figure 6 . Complex Sequential Model Summary

Model: "sequential"		
Layer (type)	Output Shape	Param #
dense (Dense)	(None, 16)	336
dense_1 (Dense)	(None, 12)	204
dense_2 (Dense)	(None, 8)	104
dense_3 (Dense)	(None, 1)	9
Total params: 653		
Trainable params: 653		
Non-trainable params: 0		
None		

Figure 7 . Simple Sequential Model Summary

Reliable Parameter Server algorithm was based on training complex neural networks; however, my hardware was limited to single CPU and single GPU. Therefore, I selected training a much more simpler Deep Neural Network model. There are two sequential model summaries in figures 6 and 7 . Model summary in figure 7 belongs to my simple model. As it can be seen from both figures, my model has way less parameters. Complex models are used for heavier tasks such as image classification, object recognition etc. Therefore, I had to select simpler task for my simple model and I selected implementing binary classification task on **Statlog (German Credit Data) Data Set** [11]. This dataset classifies people described by a set of attributes as good or bad credit risks; thus, my simpler model is trained to classify people whether they are creditable or not. In order to learn whether parameter sharing algorithm deep learning process, I simply did the followings:

- I trained multiple models outside of simulation. Half of them are trained normally (i.e.,

without parameter sharing) and the other half are trained with weight aggregation.

- In the network simulation, I run the learning process multiple times to train different numbers of nodes. I disabled parameter aggregation for half of them, in order to compare the models' accuracies.

I will plot the results in the Results part.

As the authors of RPS algorithm's paper don't clearly mentions about usage of any ACK mechanism or synchronicity of the models, I implemented the algorithms to run asynchronously.

At the starting of each training round, the component looks its received weights; if there are received weights, the component aggregates the received weights with the local weights and continue the training phase. At the end of each training round, the component shares its weights to other nodes by the help of minimum spanning tree.

4.2 Results

As I mentioned in Section 4.1.3, I compared the models in order to find out whether parameter sharing was beneficial in terms of achieving better model accuracy.

4.2.1 Multiple Model Training Out of the Network Simulation

I trained multiple models without using the network simulation. Half of them was trained without any weight aggregation, and the other half was trained with weight aggregation. Weight aggregation was perfect as they used other model's parameters at each training round.

I trained 5 models at each time with the same parameters except for the parameter sharing and run this 10 times. At each running, each model is trained with epoch size 5 and it's run 40 times which means that each model run on training data 200 times per round. Its result as follows:

Run	1	2	3	4	5	6	7	8	9	10	Tot. Avg.
Avg. wo. aggregation	0.71	0.72	0.73	0.74	0.66	0.73	0.75	0.67	0.71	0.73	0.72
Avg. w. aggregation	0.70	0.72	0.78	0.72	0.75	0.78	0.78	0.70	0.78	0.74	0.75

Table 1. Multiple Model Training Out of the Network Simulation

I can say that parameter sharing generally improves achieved best accuracy. I observed that the model's best accuracies become same when the weights are aggregated together. I also observed that when the weights are aggregated, the model quickly reaches its best model's accuracy than the models without parameter sharing.

4.2.2 Multiple Model Training in the Network Simulation

I run the whole simulation in 3 different topology settings (5, 10 and 15 nodes in the network) 5 times. In each topology setting, the epoch was 5 and each node trained for 40 rounds which means each model run on training data 200 times per round per simulations. Its result as follows:

Setting	5 Nodes	10 Nodes	15 Nodes	Total Average
Avg. wo. aggregation	0.75424	0.74940	0.74817	0.74960
Avg. w. aggregation	0.76145	0.75568	0.75389	0.75560

Table 2. Multiple Model Training in the Network Simulation

Even in this trials, parameter sharing generally improves achieved best accuracy. However, the improvements are not good as the ones within the perfect aggregation.

4.3 Discussion

I do think that my implementation of constructing minimum spanning tree can be developed further in a such way that there will be multiple activated nodes in the step k as currently there is only one node. Yet, minimum spanning tree merging algorithms must be implemented in order to achieve that. Another improvement for the algorithm might be its adaptation to dynamic environments.

MST component selects which node will be activated based on the inactive nodes' path cost and hop count. For example, the last activated node will select the inactive node with the lowest path cost and in cases there exist multiple inactive nodes with same lowest path cost, it will select the one with the minimum hop count. This activated node selection mechanism can be improved in terms of ad hoc network's qualities. For example, hop count might be the primary selection criteria, the selection mechanism can be functionalized, or even another Machine Learning or Deep Learning model can be deployed for selecting the next inactive node in dynamic environments.

My results show that parameter sharing in deep neural network trainings can be beneficial in terms of achieving better model accuracy. For my training case, only this is not enough for improving the accuracy because there are numerous things can be done for improving the accuracy. For example, much more complex model can be tuned and used for training, dataset can be expanded for making the learning process better. Instead of sharing every weights, we can share only the weights that achieve best score (i.e., accuracy on the data). Therefore, there are lots of thing to try in order to make a better model.

Another result that I observed from the result section was that models reach their best accuracy quicker when they share their parameters. This can be used for reducing training rounds as the models will reach their best condition faster. Reducing training rounds results in less network usage.

In addition to achieving better results with parameter sharing, this approach can also be useful in terms of reducing communication cost. I implemented the MST algorithm before running the RPS algorithm in order to reduce communication cost as every weight will be shared with the lowest possible costs to all nodes.

5 Conclusion

In this project, I have implemented two distributed algorithms. The first one was Anytime Distributed Minimum Spanning Tree algorithm and the second one was Reliable Parameter Server algorithm. The minimum spanning tree algorithm is designed for fixed networks in which each node knows the whole topology; however, I implemented the algorithm without the nodes knowing the topology as they first discover their neighbors and start distributively

construct the tree. The second algorithm was tested for training complex models by the authors, and I also tested it on much more simpler model. The results show that the RPS algorithm can be used for achieving better models in more quicker time. It can be used for reducing network usage. I used minimum spanning tree for parameter sharing in the learning process in order to reduce communication cost.

My implementations can be further improved. Minimum spanning tree algorithm can be adapted to be used in dynamic networks and also it can be further parallelized. Sequential Deep Learning model can be tuned further for achieving better results on training data.

References

- [1] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, “A survey on distributed machine learning,” *CoRR*, vol. abs/1912.09789, 2019. [Online]. Available: <http://arxiv.org/abs/1912.09789>
- [2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge computing: Vision and challenges,” *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.
- [3] W. Y. B. Lim, N. C. Luong, D. T. Hoang, Y. Jiao, Y. C. Liang, Q. Yang, D. Niyato, and C. Miao, “Federated learning in mobile edge networks: A comprehensive survey,” *IEEE Communications Surveys Tutorials*, vol. 22, no. 3, pp. 2031–2063, 2020.
- [4] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997.
- [5] E. Alpaydin, *Introduction to Machine Learning*, 2nd ed. The MIT Press, 2010.
- [6] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.
- [7] G. Trigeorgis, F. Ringeval, R. Brueckner, E. Marchi, M. Nicolaou, B. Schuller, and S. Zafeiriou, “Adieu features? end-to-end speech emotion recognition using a deep convolutional recurrent network,” 03 2016, pp. 5200–5204.
- [8] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015. [Online]. Available: <https://doi.org/10.1038/nature14539>
- [9] O. Arslan and D. Koditschek, “A recursive, distributed minimum spanning tree algorithm for mobile ad hoc networks,” 07 2014.
- [10] C. Yu, H. Tang, C. Renggli, S. Kassing, A. Singla, D. Alistarh, C. Zhang, and J. Liu, “Distributed learning over unreliable networks,” 2019.
- [11] D. Dua and C. Graff, “UCI machine learning repository,” 2017. [Online]. Available: <http://archive.ics.uci.edu/ml>