Scalable Computing Systems Laboratory

École Polytechnique Fédérale de Lausanne

To share or not to share? What is the cost of privacy?

By youssef el ouazzani

Project Report

Professor Anne-Marie Kermarrec

Project Advisor

Dr. Rafael Pires

Project Supervisor

EPFL IC IINFCOM SaCS

BC Building, Office BC 347

CH-1015 Lausanne

Switzerland

11-06-2021

Chapter 1

Introduction :

Federated learning is a machine learning technique that trains an algorithm across multiple decentralized edge devices or servers holding local data samples, without exchanging them. This absence of raw data movement made this machine learning technique gain a lot of traction as a privacy-preserving distributed machine learning architecture. This approach stands in contrast to traditional centralized machine learning techniques where all the local datasets are uploaded to one server, as well as to the more classical decentralized approaches, which often assume that local data samples are identically distributed.

Removing the raw data movement allows central entities (combining user data into machine learning models) to only have access to processed data, whereas users locally guard their raw private information.

But what would happen if the nodes start sharing a part of their data with their neighbours, and use this data in the training process? how would this sharing affect the results obtained?

That pushes us to ask the question: Should we share or not share? And thus, what is the cost of privacy?

In this project we use an already implemented API for decentralised machine learning, based on a D-PSGD algorithm (gossip model) using pytorch, to add an implementation of a Classification model (which was in our case the logistic regression machine learning model) and then the sharing of data during the execution between the different nodes and their neighbours. Our study will be based on using different parameters to see the effect of the sharing and have a clear comparison of the different executions at the end.

Chapter 2

Background:

For this project, we will need to first define an algorithm for the decentralised learning we will be using, which is as specified in the introduction D-PSGD (Decentralized Parallel Stochastic Gradient Descent). To define D-PSGD, we need to first determine what a stochastic Gradient Descent technique rely on.

Given a loss function, gradient descent attempts to solve the optimization problem (finding the value of x such that the loss is minimum, this is done by updating the parameter x until we reach a stable state $x \leftarrow x - \gamma \Delta f(x)$ where y is the learning rate). In stochastic gradient descent (SGD) the gradient is instead computed, and the parameters updated one-by-one for each data point. In addition, the dataset is shuffled once we have gone through all data points. SGD generally performs better at avoiding local minima than simple gradient descent.

Decentralized SGD is the application of the above technique in a decentralized setting. This will rely on separating the data to different nodes (here we only consider the case where each node's data is independently and identically distributed) with the constraint that a node only has access to its local data and can only communicate its model weights to its neighbours depending on the network topology chosen.

After defining the algorithm used, we need to choose a machine learning model we will be using to train our data. We chose to go with a Logistic regression model which is a statistical linear model that predicts a dependent data variable by analysing the relationship between one or more existing independent variables. For example, a logistic regression could be used to predict whether a political candidate will win or lose an election or whether a high school student will be admitted to a particular college. The resulting analytical model can take into consideration multiple input criteria.

Then finally after choosing this model, we must find a dataset that we would work with, and in our case, we will go with MNIST dataset. It consists of 28 * 28 pixel Gray-scale images of handwritten digits (from 0 to 9) along with their ground-truth labels. The training dataset contains 60,000 images while the test dataset contains 10,000 images.

Chapter 3

Implementation:

The implementation started by loading the MNIST Dataset, which for more convenience we had to transform into a list of tensors so that we could easily implement the linear model we chose. Then, we had to divide the dataset between all the nodes so that each one has its own training set (as explained in the Background part).

As for the training process, we implemented it by first putting a step function that will be called each time for updating the model during the training. The main objective of this function will be to:

Given each training instance:

1. Calculate a prediction using the current values of the coefficients.
2. Calculate new coefficient values based on the error in the prediction.

The process is repeated until we go through all the training set of the node, then we move to the next epoch.

So, the algorithm we used for each training step is the following:

1. X ← next batch of samples
2. y ← ground truth-labels
3. Output ← Our_model(X)
4. Loss ← CrossEntropyLoss (output, y)
5. Loss.Backward()
6. Our_model.weights ← Our_model.weights – lr* Our_model.gradient

Notes:

-Output holds the predicted labels

-the loss is computed using a CrossEntropyLoss () method based on calculating the probabilities of events.

-the Backward function is used for Back-propagation, it accumulates the gradient for each loss value, and hence each sample, in the gradient of the network

-step 6 corresponds to the Gradient Descent, it updates the network weights which is how the network "learns" from the batch of data.

The next Step was to implement the most important part of this project, which is the sharing between the nodes. The logic behind this part is that for each node at each iteration of the algorithm, we first take a random sample of the trainset then we send it to the neighbours (according to the topology used). Upon receiving this shared data, the neighbour node first drops all the duplicate elements and only adds the values that it doesn't already have in its trainset, and after this we perform the training step as explained before.

Note:

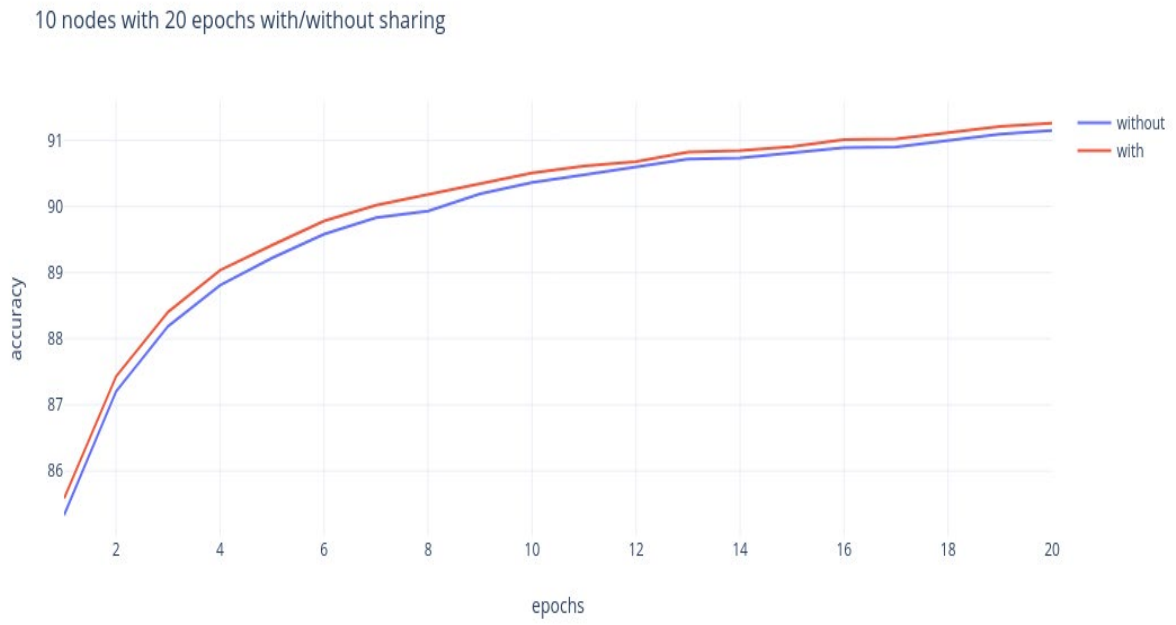
-The implementation of the sharing would have been easier if we used Pandas Data frame but due to the complexity of the data type and nature (especially the transformation we performed to Tensors while loading the data) , this approach wasn't possible. So, the first approach was to just transform the tensors to lists then send a random sample to the neighbours and back to tensors whenever we reached the training step. But one of the main problems encountered for this was the execution time, for instance this implementation of the sharing gave an execution time of 11234 seconds for 10 epochs and 4 nodes, which was very slow. So, we had to perform some optimization. The solution was to create an index table for each element of the node's trainset (the index will be calculated by combining the rank of the node to normal indexing so that each element of the whole dataset is unique and can be identified by all the nodes.), as for dropping the duplicate elements this becomes easy since we only have to iterate through the index table of the shared data and compare it with the node's trainset index table.

Chapter 4

Evaluation:

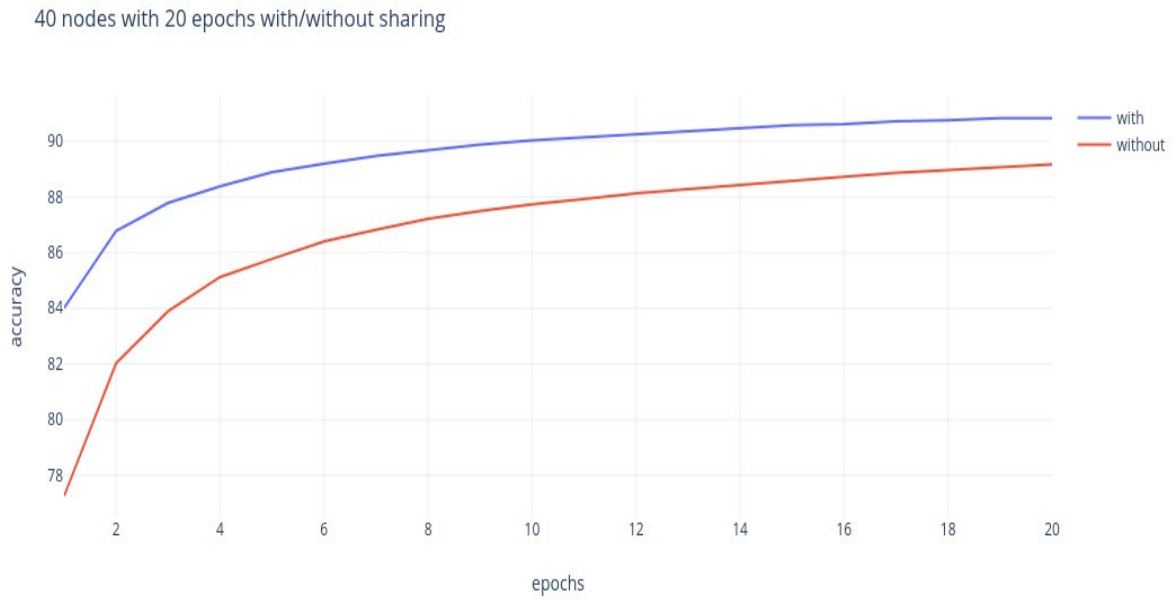
After the implementation part, we can now start doing some executions and conduct our experiments by variating the different parameters. We will first start by variating the number of nodes with a fix number of epochs and compare the two executions (with and without sharing).

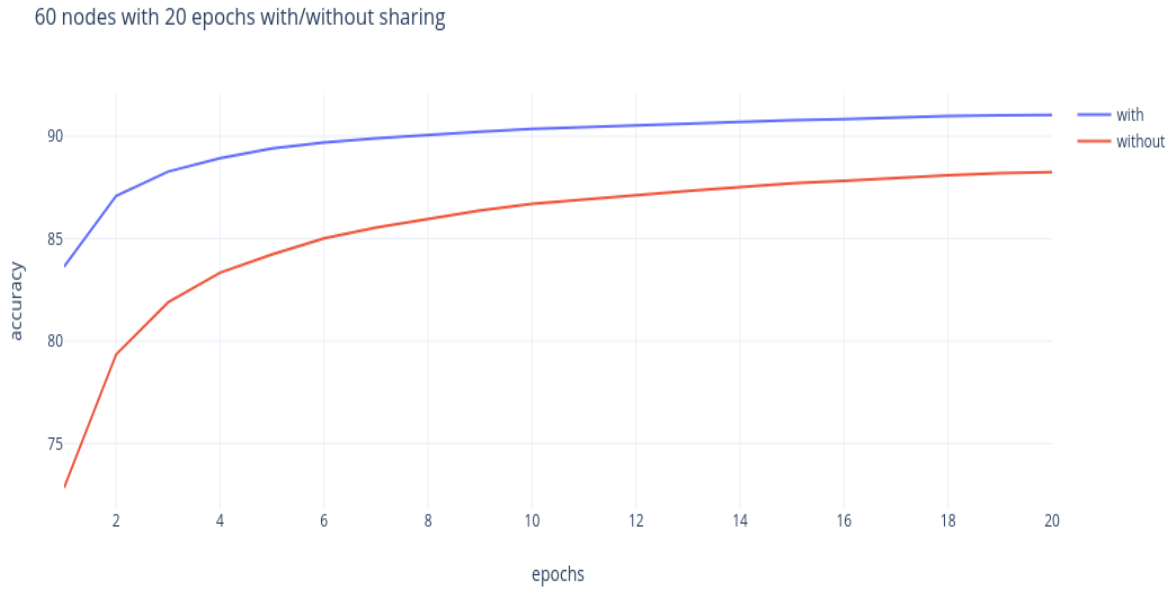
For this first experiment, we chose to go with 20 epochs and 10 nodes, here is the graph we obtain with the mean accuracy at each epoch of the 2 executions (one with sharing and the other one without):



we can clearly see that the red curve (with sharing) performs better than the blue one (without sharing) at every epoch. But what happens if we increase the number of nodes while keeping the same value of epochs.

Here is the graph of two other executions performed with respectively 40 nodes and 60 nodes:



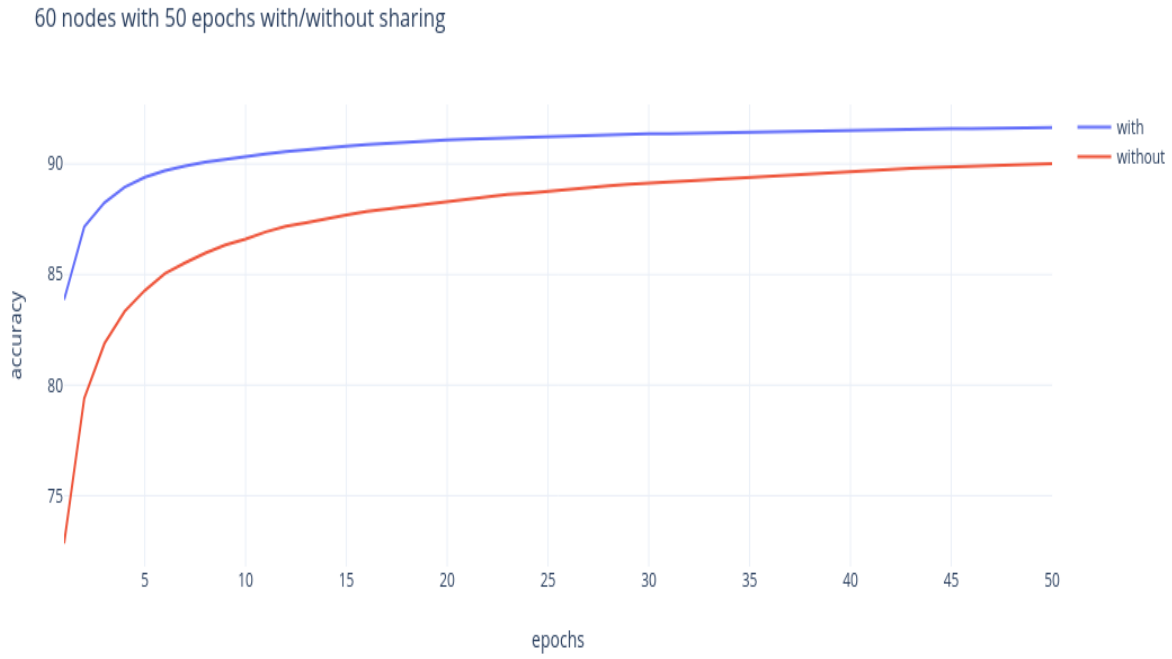


As we can see, the blue curve (with sharing) is still performing better than the red one (without sharing), but the gap in the graph between the two curves is more and more important as we increase the number of nodes. For instance, at the end of epoch 1 the difference between the 2 curves is very important (around 7% for the 40 nodes execution and 9% for the 60 nodes execution) whereas in the previous graph (20 nodes), there wasn't that much difference between the 2 curves (around 1 %), this can be explained by the fact that by adding more nodes, there is much more circulating data between them, which allows more training at every epoch and every iteration. And finally, more data for training means an increase in the accuracy since the model learns by predicting and then calculating new coefficient values based on the error in the prediction made previously.

But does this difference in performance still hold if we increase the number of epochs this time?

Well first, let's define what an epoch is, an epoch indicates the number of passes of the entire training dataset the machine learning algorithm has completed. So, the number of epochs is a hyperparameter that defines the number of times our learning algorithm worked through the entire trainset. One epoch means that each sample in the training dataset has had an opportunity to update the internal model parameters. In general, too many epochs may cause your model to over-fit the training data. So, as the number of epochs increases, we have that the weights are changing more times in the neural network and thus the curve goes from underfitting to optimal to overfitting.

In order to avoid overfitting the trainset we have at each node, we will be choosing a value of 50 epochs and 60 nodes. Here is a graph that compares both executions:



So, from the graph, we can see that increasing the number of epochs doesn't affect the fact that the sharing gives us better accuracy at every epoch than the without sharing execution, but the accuracy itself at the end of the 50 epochs is greater than the one obtained with the same parameters and 20 epochs (previous graph 20 epochs 60 nodes). So if we want to apply this sharing to real life situations, we would have to just look for the optimal value of the hyperparameter nb_e (number of epochs) so that we don't overfit the model (Note: using 50 epochs here was just an example not the optimal value).

However, all these previous executions used a ring topology, which in practice means that every node is connected to its immediate rank neighbours in the network. For instance, a node with rank r is connected to the nodes with ranks: r+1 and r-1 (mod the number of nodes)

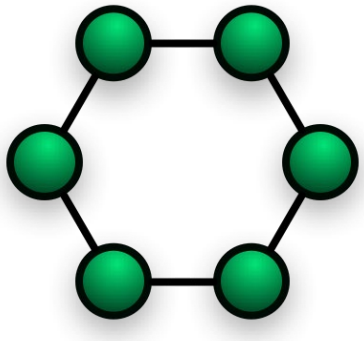
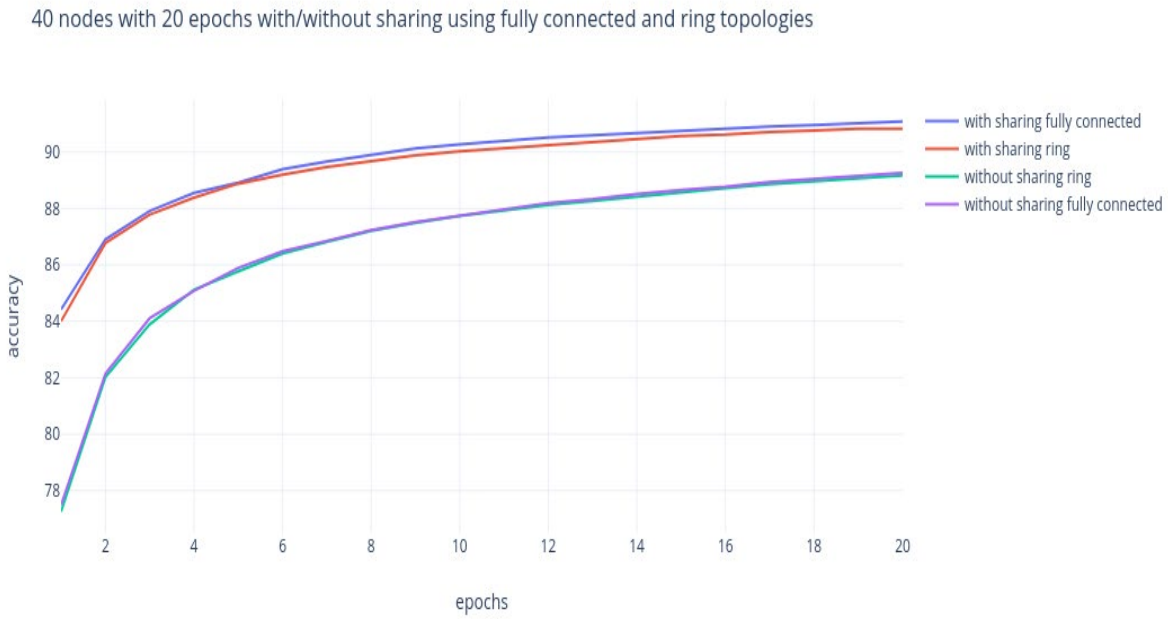


Figure 4.4 – A ring topology with 6 nodes

What would happen if we try another more intricate network topology (with more connections between the nodes), for instance fully connected topology?

So, the following graph represents 4 executions with 20 epochs and 40 nodes comparing the fully connected topology with the ring topology:



From this graph, we can clearly see that without sharing, having a fully connected or a ring topology gives us approximately the same accuracy curve. However, when we use the sharing, we can observe that the blue curve (fully connected topology) performs slightly better than the red one (ring). This difference is due to the nature of a fully connected topology (Every node is connected to all other nodes in the network).

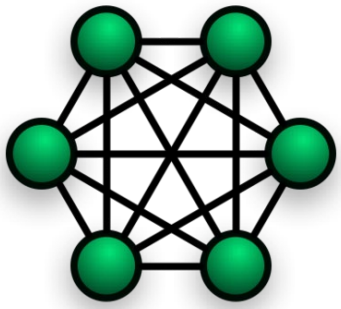


Figure 4.3 – A fully connected topology with 6 nodes

This will allow every node to train more since it receives shared data more frequently from multiple other neighbours, but this slight improvement comes with a very important time cost. Let's compare the two execution times of the blue and red curves.

result_with_e20_ps20_time.txt 18 Bytes

```
1  1476.3696703910828
```

(red curve execution time)

result_with_e20_ps40_fc_time.txt 17 Bytes

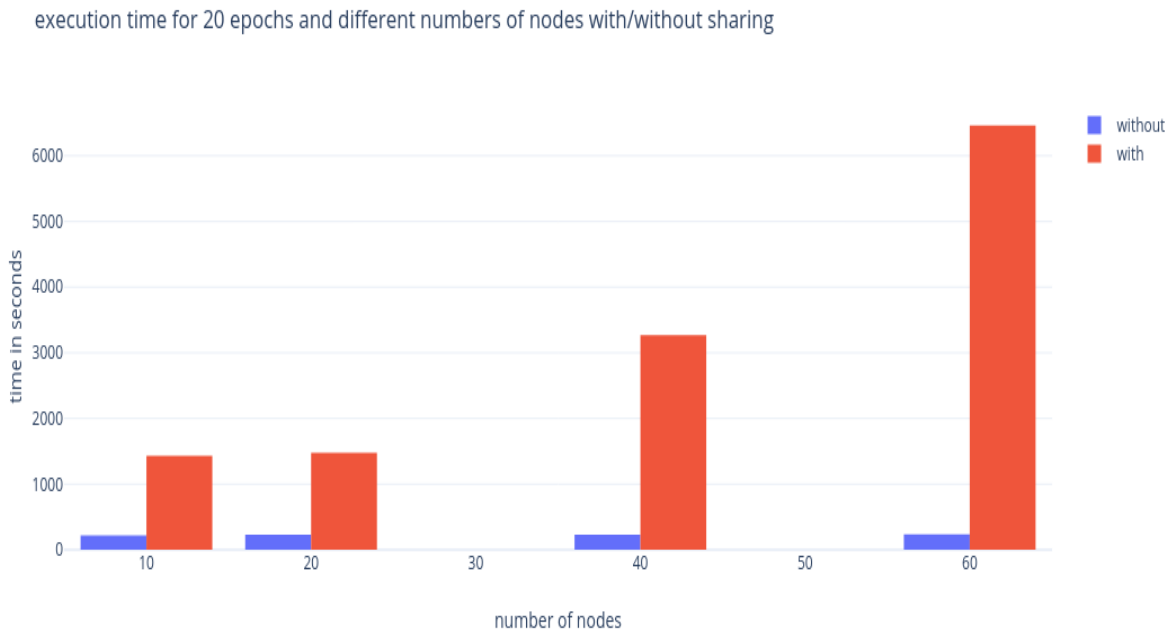
```
1  41636.22145199776
```

(blue curve execution time)

The blue curve time is approximately 30 times greater than the red curve's execution time. This happens because in a fully connected topology, there will be much more traffic while sharing, since there are much more connections between the nodes. So, is it worth it to use a fully connected topology for the sake of having this slight improvement?

This question is broad since it depends on the context of using the model and on the trade-off (time/accuracy) we want to have.

Let's now consider the previous executions but from a time point of view. If we represent all the execution times in a graph (ring topology is used for these executions):



We can clearly see that the multiple executions with sharing take a lot more time, whereas the without executions have a stable execution time (varying from 220 to 240 seconds). We also observe that the more we increase the number of nodes the more the time increases in an exponential fashion, this can be explained by the fact that sharing as stated before means more data to train with, at every node and every epoch. But there is another reason that could also explain the increase in time, it's the fact that the sharing is done by taking a random sample from every node's trainset, after this the receiving node should check for duplicates elements before adding the shared data received in its trainset, so in conclusion every time a node receives new shared data, it doesn't train with everything it receives, this results in having more and more iterations until the node receives all the circulating shared data.

Note:

One possible solution is to store the index of the elements that it shares and every time a node tries to send some data by taking a random sample, we first check if any of this data to be sent has been sent in previous iterations if yes change it by taking another element. This way, we will be sure that each time a node receives data, it is not duplicate elements and we can maybe have a decrease in the time it takes to execute the training.

Chapter 5

Conclusion:

We implemented in this project a logistic regression model and used a D-psgd algorithm to train it on an Mnist Dataset. We then added a sharing option, which relies on the fact that the nodes are connected to some neighbours according to a topology network, that allowed these nodes to have more data to train with.

After conducting multiple experiments by variating the number of nodes, epochs but also the topology network, we concluded that the sharing performs better than without sharing and especially if we increase the number of nodes. But this comes with a time cost that should be taken into consideration since it grows exponentially when we increase the number of nodes and the topology network. Thus, there will always be a trade-off (accuracy/ time) that we will face by using this implementation.

Furthermore, in a real life situation, if our classification implementation was based on users, by applying the sharing we would have multiple users on one node, so the nodes will have more information on what other users would want (depending on the neighbours, the topology used and circulating shared data) and be able to generalize this more to different users.

Bibliography

[1] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. "Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent". In: Advances in Neural Information Processing Systems. 2017, pp. 5330–5340.

[2] Tian Li, Anit Kumar Sahu, Ameet Talwalkar; Virginia Smith. "Federated Learning: Challenges, Methods, and Future Directions".

[3] Amitrajit Bose. "Handwritten Digit Recognition Using PyTorch — Intro To Neural Networks".

[4] https://www.statisticssolutions.com/what-is-logistic-regression/

[5]https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#sphx-glr-beginner-blitz-neural-networks-tutorial-py

[6] Felix Sattler, Robust and Communication-Efficient Federated Learning from Non-IID Data, arXiv:1903.02891