Computer Science
Master's Semester Project

Data sharing for tackling non-iidness

Student: Joseph Abboud

Professor: Prof. Dr. Anne-Marie Kermarrec
Assistants: Rishi Sharma
Dr. Rafael Pereira Pires

January 6, 2023

Scalable Computing Systems Laboratory
EPFL

Contents

1 Introduction

In recent years, decentralized machine learning has gained increasing attention as a machine learning paradigm that allows multiple nodes to collaborate and learn from each other without the need for a central authority [3, 17]. In decentralized learning systems, each node is responsible for learning a part of the overall model and sharing its knowledge with other nodes in the network. This distributed approach has several advantages, such as improved scalability and robustness, as the system can continue to operate even if some nodes fail or are disconnected [13].

However, decentralized learning also has several challenges, one of which is designing efficient communication and collaboration protocols that allow the nodes to effectively learn from each other while minimizing the communication overhead. This is especially important in large-scale decentralized systems, where the number of nodes can be very large and the communication resources are limited. We will tackle the issue of non-IID data, which can slow down convergence and lower rewards by causing each node to train a different model than its neighbors [7].

One approach to addressing this challenge is to use the label distribution of nodes to create more efficient topologies. In machine learning classification tasks, each data sample is usually associated with a label, which indicates the class or category to which the sample belongs. By considering the label distribution of nodes, it is possible to form connections between nodes using some effective unit of comparison, as these nodes are likely to have more relevant information to share with each other.

In this project, we investigate the use of label distributions to design efficient topologies for decentralized learning systems. We explore different methods for comparing the label distribution of nodes and evaluate their effectiveness in creating efficient topologies. We also investigate the trade-offs between communication efficiency and model performance, and study how different topologies and label distributions affect the convergence and generalization of the decentralized learning system by trying to reduce the effect of non-IIDness of the data.

The first, more experimental part of the project will try to answer the following question: **Can we use the label distribution of a node to assign to it appropriate neighbors and reduce the negative effects of non-iidness?** The second part is more open, where we showcase different algorithms that generate interesting results when taking into account label distributions.

2 Background

2.1 Machine Learning and Decentralized Learning

We assume that the reader possesses enough understanding around machine learning, deep learning [10] and more precisely federated and decentralized learning [15]. One also needs to understand the concept of iid and non-iid data, and their implications in decentralized machine learning [7].

2.2 non-IID data

There are several challenges that arise when working with non-iid data in decentralized machine learning:

1. **Data imbalance**: If some nodes have more data than others, it can lead to a data imbalance, which can negatively impact the performance of the machine learning model.
2. **Limited sample complexity**: The sample complexity of the model may be limited by the amount of data available on each node. This can make it more difficult to achieve good performance with decentralized machine learning.
3. **Heterogeneity of data**: Non-iid data can be more heterogeneous, which can make it more difficult to learn robust models.

2.3 Vector Similarity Metrics

There are various metrics that can be used to measure the similarity between two vectors. Here are the formulas and definitions for the ones used in this project:

- **Cosine similarity**: Given two vectors $\mathbf{u}$ and $\mathbf{v}$, the cosine similarity is defined as $\cos(\theta) = \frac{\mathbf{u}\cdot\mathbf{v}}{||\mathbf{u}||\cdot||\mathbf{v}||}$, where θ is the angle between the two vectors and $||\mathbf{u}||$ and $||\mathbf{v}||$ are the magnitudes of the vectors. In this project, the vectors only contain positive values, so $\theta \in [0, \frac{\pi}{2}] \implies \cos(\theta)$ is strictly decreasing in $[0, 1]$
- **Pearson correlation**: Given two vectors $\mathbf{u}$ and $\mathbf{v}$, the Pearson correlation is defined as $\rho_{\mathbf{u},\mathbf{v}} = \frac{\text{cov}(\mathbf{u},\mathbf{v})}{\sigma_{\mathbf{u}}\sigma_{\mathbf{v}}}$, where $\text{cov}(\mathbf{u}, \mathbf{v})$ is the covariance between the two vectors and $\sigma_{\mathbf{u}}$ and $\sigma_{\mathbf{v}}$ are the standard deviations of the vectors. $\rho_{\mathbf{u},\mathbf{v}} \in [-1, 1]$
- **Manhattan distance**: Given two vectors $\mathbf{u}$ and $\mathbf{v}$, the Manhattan distance is defined as $d(\mathbf{u}, \mathbf{v}) = \sum_{i=1}^{n} |u_i - v_i|$, where n is the number of dimensions of the vectors.
- **Euclidean distance**: Given two vectors $\mathbf{u}$ and $\mathbf{v}$, the Euclidean distance is defined as $d(\mathbf{u}, \mathbf{v}) = \sqrt{\sum_{i=1}^{n}(u_i - v_i)^2}$, where n is the number of dimensions of the vectors.
- **l-t distance**: Given two vectors $\mathbf{u}$ and $\mathbf{v}$, the l-t distance is a generalization of the two previous distances. It is defined as $d(\mathbf{u}, \mathbf{v}) = \sqrt[t]{\sum_{i=1}^{n} |u_i - v_i|^t}$, where n is the number of dimensions of the vectors.
- **Kullback-Leibler divergence**: Given two vectors $\mathbf{u}$ and $\mathbf{v}$, the KL-divergence is defined as $KL(\mathbf{u}||\mathbf{v}) = \sum_{i=1}^{n} u_i \log\left(\frac{u_i}{v_i}\right)$ where n is the number of dimensions of the vectors. It is a measure of the difference between two probability distributions, and is used in machine learning and statistics to compare the similarity between two distributions, with lower values indicating greater similarity.

2.4 Basic Graph Theory

This project requires some understanding of basic notions in graph theory. A graph is a collection of vertices (also known as nodes) and edges that connect them. The vertices represent the entities being modeled, and the edges represent the relationships between them. Graphs can be either directed or undirected. We will entirely work with undirected graphs here.

There are several types of graphs that are commonly studied in graph theory, and that are used in this project as well. A d-regular graph is a graph in which each vertex has exactly d edges incident to it. A ring is a graph that consists of a single cycle, and a fully connected graph is a graph in which every pair of vertices is connected by an edge.

Graph theory is the study of these structures. It includes algorithms to build them, traverse them, sparsify them etc. This project will use graph search algorithms, notably Breadth First Search [1].

3 DecentralizePy

Note: For a more thorough explanation of the framework, refer to a Master's thesis done last year by Jeffrey Wigger at SaCS. [18].

DecentralizePy [4] is a framework written in Python 3 used for running distributed Machine Learning applications. It is being maintained by the SaCS lab at EPFL. The framework is partitioned into modules that can easily be swapped with other modules. It can run on a single machine as well as on multiple servers. In both cases, it runs one to many processes per machine, each representing a node in distributed machine learning. The different modules of the framework can be seen in the diagram in Figure 1. These are replaceable by more specific classes (e.g., Node can be replaced by DPSGDNode or PeerSampler etc.). In the following, we will talk more about each module, and explain in detail the classes that were used, changed or added in the scope of this project.

3.1 Repository

The repository is publicly available on EPFL's Gitlab. It is divided in 3 directories:

- Firstly, `eval`, the evaluation directory, contains different config files (`config*.ini`), graph topologies (`*.edges`), server IPV4 addresses (`ip*.json`), plotting scripts (`plot*.py`), main Python scripts (`testing*.py`) and Bash scripts used to run a main script with the correct config file and arguments (`run*.sh`).
- The second directory, `tutorial`, is more minor: it contains a copy of each type of file from `eval` and is used as a more convenient way to run the framework. *Note:* the Bash script needs to be run on each machine.
- The `src/decentralizepy` directory contains the source code. It contains one directory for each module as shown in Figure 1 and the following subsections. Any changes or additions to the `src` directory will be shown in 4.

3.1.1 Communication

The `Communication` class is an API for communication between processes in DecentralizePy. It has several methods for encoding and decoding data, connecting and disconnecting with neighbors, and sending and receiving messages. When an instance of the Communication class is created, it is initialized with a rank, machine ID, mapping, and total number of processes. The subclass used in the project is `TCP`.

TCP The `TCP` class is a subclass of `Communication`, and provides an implementation of the communication API using the Transmission Control Protocol (TCP) and the ZeroMQ (ZMQ) library [6]. It has several additional methods for initializing and closing connections with neighbors, as well as for sending and receiving data over TCP. When an instance of the `TCP` class is created, it is initialized with a rank, machine ID, mapping, total number of processes, and the filepath to a JSON file containing a mapping of machine IDs to IP addresses. Every process is uniquely identified in the communication protocol by a port number, determined by its unique rank (see 3.1.5 to understand rank, machineID and UID).

- The `encrypt` method in this class encodes data as a Python pickle object, while the `decrypt` method decodes received pickle data.
- `init_connection` initiates a connection to a given neighbor, `close_connection` closes a connection with a neighbor, `connect_neighbors` establishes connections with all neighbors, and the `disconnect_neighbors` method terminates all connections with neighbors.

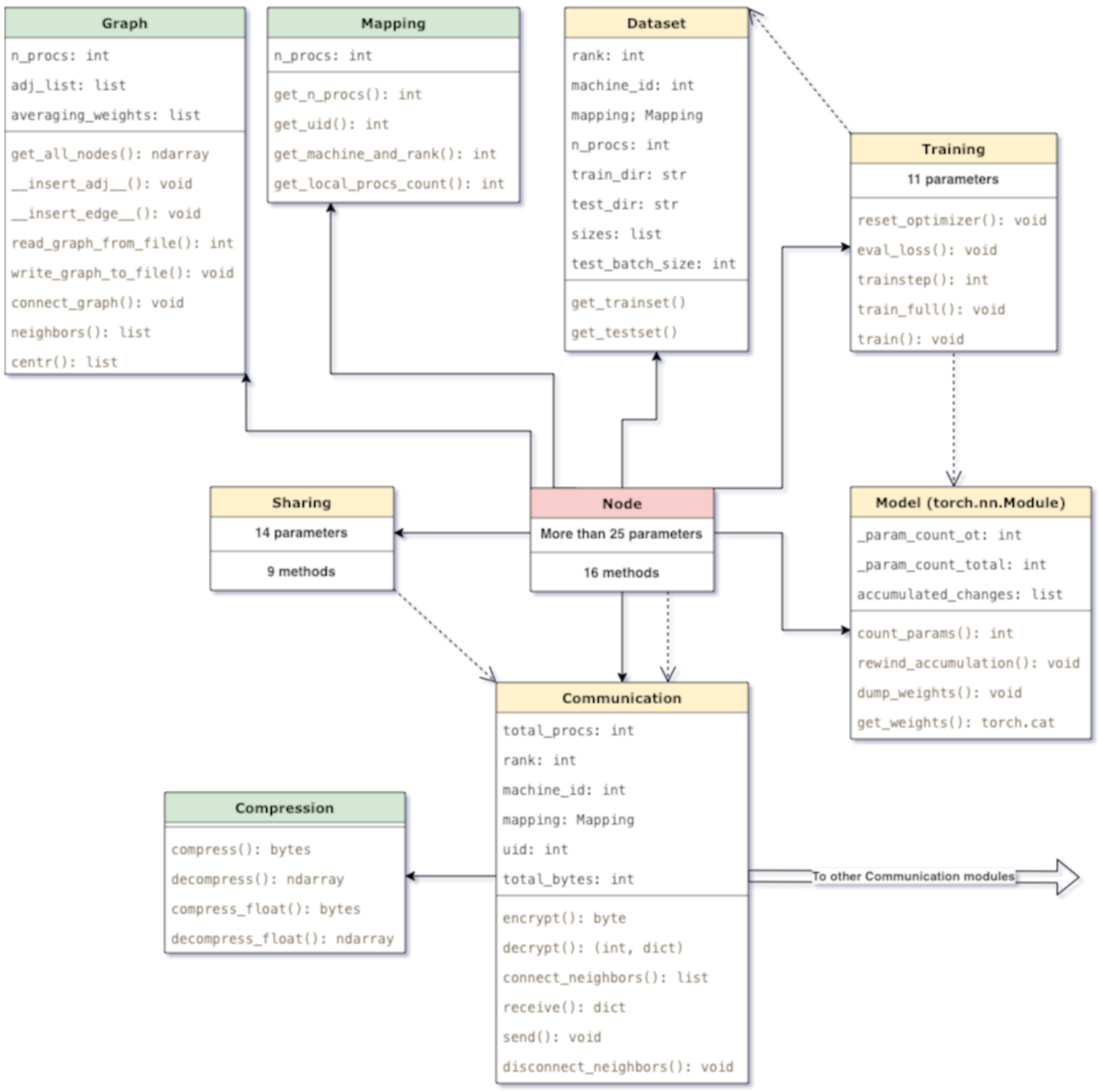

Figure 1: Architecture of the DecentralizePy framework centered around a single node. Nodes are connected via the Communication module. A full arrow from module A to module B means that A has a parameter of type B. A dotted arrow means that module A's methods are called by module B. Red modules are primary, the yellow ones are secondary, and the green ones are used as utils.

- The `receive` method receives data from a connected neighbor, while the `send` method sends data to a specific neighbor.

3.1.2 Compression

The `Compression` class is an API for compressing and decompressing data in DecentralizePy when communicating between processes.

- The `compress` method takes in an array of data (usually array of int) and returns the encoded data as a `bytearray`.
- The `decompress` method takes in compressed data as a `bytearray` and returns the decompressed data as an array.
- `compress_float` and `decompress_float` are the same as above, but more specific to float arrays. They are used for the compression of the model's floating point parameters.

3.1.3 Datasets

The `Dataset` class is an API for working with datasets in DecentralizePy. It has two methods, `get_trainset` and `get_testset`, that are intended to return the training and test sets, respectively. It is initialized with a rank, machine ID, mapping, and several optional arguments for specifying the directories where the training and test data are stored, the sizes of the partitions for each process, and the batch size for testing. There are many dataset subclasses written for the framework (FEMNIST [2], Celeba[14]...), but we will talk in detail about CIFAR-10 [9], since it is the only dataset used in this project.

There is also a simple `Data` class in this module, for storing data samples and their corresponding labels. It has a constructor that takes in the data samples and labels as `numpy` arrays, as well as two special methods, `__len__` and `__getitem__`, that are required for it to be used as a data object in PyTorch. `__len__` returns the number of samples in the dataset, while `__getitem__` returns the data sample and label at a given index.

CIFAR-10 The CIFAR-10 dataset [9, 16] was developed by researchers at the University of Toronto. It is a popular dataset for image classification, which consists of 50,000 32x32 color training images and 10,000 test images. The images are labeled with one of 10 classes: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The dataset is divided into 5 training batches and 1 test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Overall, the dataset is well-balanced with respect to the number of images in each class

`CIFAR10.py` contains CIFAR10, a subclass of `Dataset`. It is specific to CIFAR-10 and provides a way to load, partition and access the training and testing sets for this dataset. The constructor takes several arguments including the rank and machine ID of the current process, a Mapping object that provides a mapping between ranks and machine IDs to unique IDs, the directory where the training and testing data is stored, the sizes of the partition for each process, the batch size for testing, and a boolean flag indicating whether to partition the data in a non-IID way (see later for more info on IID). The CIFAR10 class also includes a transform attribute which specifies how to preprocess the data before it is used for training or testing.

GN-LeNet The model used with CIFAR-10 in this project is called `LeNet`, and is inspired from the original LeNet [11] for MNIST. It is a convolutional neural network (CNN) [12] with three convolutional layers, each followed by a max pooling layer and a group normalization layer. The first convolutional layer has 32 filters of size 5x5 and takes as input 3-channel images (since

CIFAR-10 consists of color images). The output is then passed through a max pooling layer, which downsamples the output by a factor of 2 in each dimension, and a group normalization layer, which normalizes the activations within each group of filters.

The second and third convolutional layers have the same architecture as the first, but the number of filters is increased to 32 and 64, respectively. The output of the third convolutional layer is passed through another max pooling layer and group normalization layer before being flattened and fed into a fully connected (FC) layer with 10 output units, corresponding to the 10 classes in CIFAR-10. The output of the FC layer is the final classification logits of the model.

There are $89,578$ trainable parameters in total (including biases):

1. Conv1: $3 * 32 * 5^2 + 32 = 2,432$
2. Conv2: $32 * 32 * 5^2 + 32 = 25,632$
3. Conv3: $64 * 32 * 5^2 + 64 = 51,264$
4. FC: $64 * 4^2 * 10 + 10 = 10,250$

3.1.4 Graphs

The `Graph` class defines a graph topology and provides functions to read and write the graph to/from a file. The graph is represented as an adjacency list of sets of vertices. The class has a constructor that takes an optional parameter `n_procs`, the number of processes in the graph. It also has a `get_all_nodes` function that returns a list of all nodes in the graph. The class has functions to insert edges or adjacencies into the graph. It also has a function to connect the graph using a ring, which connects each vertex to its two nearest neighbors. In the project, unless stated otherwise, the initial topology always contains 96 processes and is either a fully connected or a 4-regular graph.

3.1.5 Mappings

The `Mapping` class is an abstract base class that defines a bidirectional mapping between a unique identifier (uid) and the machine id and rank of a process. It has methods to retrieve the number of global processes, the uid given a rank and machine id, the machine id and rank given a uid, and the number of node-local processes. In the project, the mapping used is linear, via the formula:

$$uid = machine\_id * procs\_per\_machine + rank \tag{1}$$

`Linear` is a subclass of `Mapping`, with an additional parameter, `global_service_machine`, which specifies the machine that serves as the host for special nodes: at the global service machine, the uid of a special node can be a negative number (for example, see `PeerSampler` at 3.1.7).

3.1.6 Models

The `Model` class is a wrapper for a PyTorch neural network model. It has several attributes that store information about the model's parameters, such as the number of trainable parameters and the accumulated changes. It also has several methods for interacting with the model's parameters, such as counting the total number of parameters, resetting the accumulated changes, dumping the weights to a file, and flattening the current weights. It is the parent class to `LeNet`, the CNN used in the project (see 3.1.3).

3.1.7 Node

This is the core of DecentralizePy. There are different types of `Node`s used in the project. The hierarchy of these classes is shown in Figure 2. The `Node` class represents a node in a decentralized machine learning framework. In general, there are as many `Node` instances as there are vertices in the topology. As we see in Figure 1, this is where the bulk of the work happens when the code is running. A `Node` will use the other modules to train its model, share its parameters, communicate with other nodes etc. It has methods for connecting and disconnecting with neighbors, sending and receiving messages through a communication channel. It also has a message queue for storing received messages, a set for tracking connected neighbors, and a unique identifier for the node. The `Node` class can connect/disconnect to its neighbors by sending a connection/disconnection request and waiting for a response. It also has methods for exchanging messages with other nodes on specific channels via channel labels (e.g., `CONNECT`, `HELLO`, `DISCONNECT` etc.). Finally, the Node class has a method for initializing a log file for logging progress, messages or any other useful information. This class is still a bit abstract. It cannot do much without defining a more concrete subclass, such as `DPSGDNode` or `PeerSampler`.

DPSGPNode The `DPSGDNode` class is a subclass of `Node` and is used for decentralized parallel stochastic gradient descent (DPSGD) [13]. It receives messages on the `DPSGD` channel. The `run` method here is the most important of the whole framework, as this is where the decentralized learning happens. This method trains the model, sends and receives messages with neighbors, and updates the model based on the received messages. `run` has to be called by child-classes of `DPSGDNode`. Finally, `DPSGDNode` has a method for saving results to a JSON file and a method for saving the model's weights. These JSON files will be used to showcase our results: they contain the training/testing losses and testing accuracy per epoch, and other useful metadata.

PeerSampler The `PeerSampler` class is another subclass of `Node` and is used, as the name suggests, for peer sampling in the framework. This is one of the "special" nodes with a negative rank (usually -1), and it resides on the global service machine. The Peer Sampler first connects to all other nodes in the topology. It then keeps running an infinite `while` loop, waiting for messages from these nodes. The loop terminates when all nodes have disconnected. In this loop, the Peer Sampler awaits `REQUEST_NEIGHBORS` messages from nodes and answers back on the `Peers` channel with a `set` containing the neighbors of the requesting node, following the graph given as an argument in the beginning. This is done in the `get_neighbors` method.

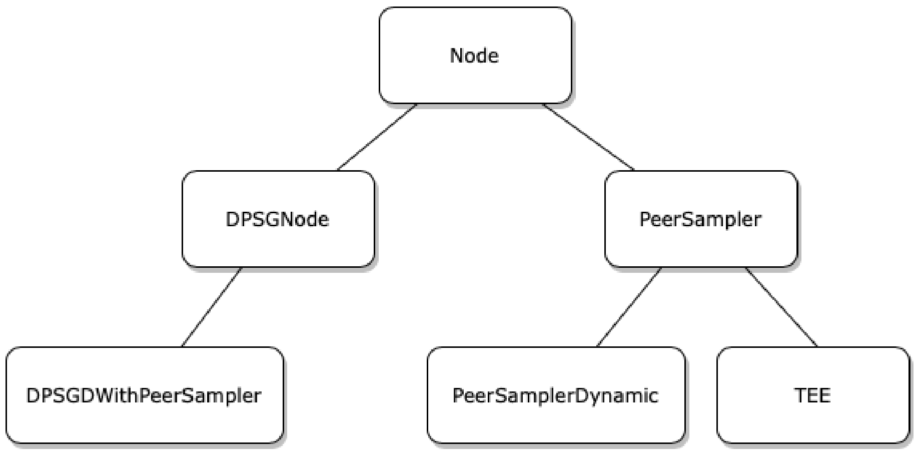

Figure 2: Hierarchy of the Node class

PeerSamplerDynamic This subclass of `PeerSampler` acts the same, except for the `get_neighbors` method. In this class, the set of neighbors returned changes after every iteration.

DPSGDWithPeerSampler Finally, this is a class that extends the `DPSGDNode`. The only difference is when getting the neighbors: a `DPSGDNode` will return its adjacency list from the initial graph, whereas a `DPSGDWithPeerSampler` will ask the Peer Sampler for its neighbors via the `REQUEST_NEIGHBORS` channel.

3.1.8 Sharing

The `Sharing` class has functions for serializing and deserializing the model, compressing and decompressing data, and sharing model parameters with other nodes in the network. It also maintains information about the rank and machine ID of the current node, the communication module used to send and receive messages, the mapping of node IDs to ranks and machine IDs, the graph representing the network topology, the model being trained, and the dataset being used. The `Sharing` class uses the `compress` attribute to determine whether or not to compress data when sharing model parameters. It also has an optional `compression_package` and `compression_class` argument that allow the user to specify a custom compressor to be used.

3.1.9 Training

The `Training` class is a module for training a neural network on a single node. It has methods for training a model on a dataset, evaluating the loss on the training set, and resetting the optimizer. It also has a method for training a model on a full epoch of a dataset and a method for performing a single training step on a minibatch of data. The class takes as input the rank and machine ID of the node, a mapping object for translating between rank and unique identifier, the model to be trained, an optimizer for learning the model's parameters, a loss function to use during training, and various parameters for controlling the training process such as the number of rounds, whether to use full epochs or minibatches, the batch size, and whether to shuffle the dataset.

4 Project evolution

In this section, I will first explain how the project developed over time from week 1 of the semester until the time of writing this report. I will then talk about each added file to DecentralizePy and explain how they work and what their role is in the framework. All the added code is publicly available on EPFL's Gitlab [8].

4.1 Timeline

4.1.1 First steps and design

At first, after understanding the code and running some experiments, my goal was to transmit the label distribution of each node to the Peer Sampler. This was implemented in `DPSGDWithPeerSampler`, in a method called `send_labels`. It is very simple: the node goes through its local dataset and counts how many data points of each class it possesses in its training set. It then sends through the `LABEL_DIST` channel a `dict` of these counts to the Peer Sampler. This method is called at initialization, before calling the `run` method of `DPSGDNode`. On the Peer Sampler's side, it waits to receive the labels from all nodes, and then calls the `run` method.

Then, two routes presented themselves to continue the project:

1. We would either build a hardware enclave in the style of Rex [5] where we store the label distributions with privacy and confidentiality.

2. The second route (which is the one we took) is to find a way to use these distributions to assign better neighbors to each node in order to potentially get more useful results.

However, we assume here that there is a trusted execution environment (TEE), and that the Peer Sampler cannot access the label distributions, which is why I called the class for this project `TEE`. Also, any code that directly manipulates the label distributions is written in C and C++.

This TEE was first of all just an interface between the Peer Sampler and the enclave. It was supposed to serve as an API that the Peer Sampler calls to transmit encrypted messages from the nodes to the enclave or to query the enclave when comparing two nodes' label distributions. So `TEE` was just a sister class of `PeerSampler` (direct child of `Node`) and would have rank -2. This turned out to be too complicated, and a simpler much more efficient solution presented itself: I made `TEE` a child of `PeerSampler` (refer back to Figure 2). This way, the enclave is located inside that class and its methods are called using the `ctypes` built-in Python library. In principle, `TEE` relays an encrypted label distribution from a node to the enclave, which is then decrypted and stored securely. The enclave then can quantify how similar two nodes are with regards to their label distribution. The peer sampling service only knows about these similarities and then determines the neighbors of the querying node (see Figure 3).

Note: as mentioned above, the enclave has **not** been built. So for now on, anything happening on the C/C++ side is considered as "enclave" and abstract to the `TEE` python class. We will refer to the latter as TEE, even though it is not really a trusted execution environment *per se.*

4.1.2 Experiments

Now that the structure has been laid, I had to figure out what to do with the label distributions. A method was added in the enclave, `similarity`, which takes as arguments two nodes and returns a `double` representing the similarity of these nodes' label distributions, given one of the vector similarity metrics defined in `similarity.cpp`. This is the only method exposed by the enclave at the time of writing (there are `print_*` methods but they are only used for testing and debugging purposes). At the start of the program, what happens at `TEE` is shown in Algorithm

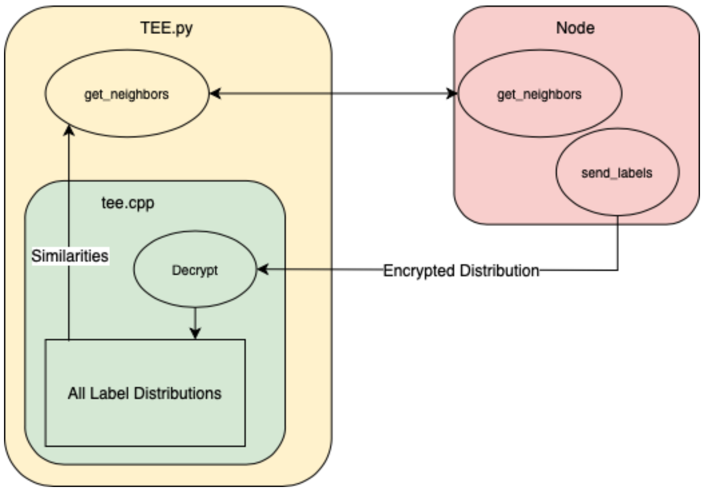

Figure 3: Diagram of TEE.py showing how `TEE` answers the `get_neighbors` query from a Node. The enclave is within the class and the labels are not reachable.

1. I then wanted to see if `TEE` can mitigate the effects on non-iidness, so I ran the new framework with different similarity metrics and neighbor orderings, and comparing them to running the framework on some random graph (see more details at 6).

Algorithm 1 Lifetime of the `TEE` node

Require: There are N training nodes in the topology. `TEE` initializes an empty table S of size $N \times (N-1)$, where $S[i]$ is a sorted list of similarities between node i and all other nodes, $i \in [N]$

```
 1: procedure COMPUTESIMILARITIES()                          ▷ $O(n^2 \log n)$
 2:     for $i \leftarrow 0$ to $N-1$ do
 3:         for $j \leftarrow 0$ to $N-1$ do
 4:             if $i \neq j$ then
 5:                 $s \leftarrow similarity(i, j)$                ▷ Query the enclave
 6:                 Append tuple $(j, s)$ to $S[i]$
 7:                 Sort $S[i]$ on $s$       ▷ Order depends on experiment, $O(n \log n)$
 8:
 9: while Not all nodes sent their labels do
10:     Wait
11: Store labels in enclave                                  ▷ Will explain in 4.2.2
12: COMPUTESIMILARITIES()
13: Build neighbors topology     ▷ Many algorithms have been used, will explain them later
14: while Not all nodes disconnected do
15:     Wait for get_neighbors queries
16:     Answer the queries from the built topology.
17: Terminate TEE
```

I tried to leverage the results of the previous experiments to get rid of data heterogeneity by devising strategies and algorithms where we group nodes into pseudo-iid clusters: this means that grouping the non-iid local datasets of these nodes yields a more or less balanced and uniform dataset, while keeping in mind communication overhead and the average amount of neighbors per node. My goal was to achieve results as close as possible to an iid framework, and maybe even exceed them.

4.2 Code

This section will quickly go over the 3 scripts that I added to the framework, so the reader can have a better understanding of what their methods do exactly.

4.2.1 C++ interface

tee.cpp As has already been mentioned, this script represents the enclave where the label distributions are supposedly securely stored. The structure of the enclave is simple and straightforward. There are three C-style `struct`s nested into each other:

1. `data`: it represents one tuple `(label, count)`
2. `node`: it represents a `DPSGDNode`. This struct is characterized by a node ID and an array of `data structs` (*Note:* this array is of size 10 with CIFAR-10 for example, 1 `data` for each class).
3. `tee`: this `struct` is unique and it encapsulates all nodes of the framework via an array of `node`. The size of the array is determined by `numNodes`.

Note: There are no arrays structure in C. What is meant by "array" above is a pointer to the first element of this virtual list accompanied by a size so we know when to stop iterating after the first pointer.

It has already been said what the `similarity` method does, but more precisely, it first checks that the 2 input nodes have the same amount of labels. Then it transforms the C `struct node` into a C++ `vector` in order to call one of the methods imported from `similarity.cpp`.

To use the C interface in `TEE`, one would need to compile this code into a shared library and then use a Python wrapper (`ctypes` in this case) to call the functions from Python. The compilation into a library has been added at the beginning of the Bash script that launches the framework.

similarity.cpp This code contains several similarity metrics between two C++ vectors. There are also other helper methods such as checking the validity of 2 vectors, computing their dot product, normalizing a vector and transforming a vector of integers (label counts) into a vector of frequencies ($count/total$). The similarity metrics defined at the time of writing are (refer to 2 for the mathematical formulae):

- Cosine similarity
- l-t distance , $t > 0$, $t \in \mathbb{N}$
- Pearson correlation
- Kullback-Leibler divergence

One can add any other similarity metric to this file, and replace the called method in `tee.cpp/similarity`.

4.2.2 TEE

We already know how this class works (see 4.1). Moreover, there are also three class variables (one for each `struct` in `tee.cpp`) that inherit from the `ctypes.Structure` class. Also, at initialization, `TEE` loads the shared library and create an empty `ENCLAVE` class, which is then filled `NODE` by `NODE` after receiving all labels. The most important argument fed to `TEE` is `top_neighbors`, which is the number of neighbors the TEE will aim to reach for each node.

Finally, we see at line 13 of Algorithm 1, that there are many algorithms used to build the topology. There are 2 methods that achieve this purpose: `build_kreg_table` (used for the experiments at 6) and `compute_neighbors_table` (used later at 7). Both methods will be explained in more detail at their respective sections.

5 Setup

5.1 Training and Testing Setup

In all the following sections, the models have been trained using the DPSGD optimization method. In order to compare the results with each other, the same dataset, hyperparameters and seeds were used with every run.

Here are all the immutable values used across all runs:

- **Dataset:** `CIFAR-10`
- **Model:** `LeNet`
- **Initial seed:** `90`
- **Is data non-iid:** `True`
- **Shards:** `4`
- **Learning rate:** `0.02`
- **Communication rounds in a global epoch:** `4`
- **SGD batch size:** `32`
- **Loss function:** `Cross entropy loss`
- **Machines:** `3`
- **Total nodes:** `96`
- **Procs per machine:** `32`
- **Iterations:** `2000`
- **Test after:** every `20` iterations first, then every `40` iterations after `1000` iterations.
- **Initial topology:** Random 4-regular graph with 96 vertices

Unless stated otherwise, all the results presented in this project used these parameters. Also, we test after every 20 epochs instead of after every one because node local testing is slow, since every node needs to run the evaluation on the testset.

5.2 Hardware Setup

All experiments were run on three similar machines with CPU:

- Intel(R) Xeon(R) E-2288G CPU clocked at 3.70GHz (16 cores, 2 threads per core) with 64GB of RAM.

6 Experiments

This section will showcase the results for the first part of the project, and try to answer the questions: **Can we use a similarity metric between nodes' label distributions to tackle non-iidness? If so, what is a good similarity measure?** In order to find an answer, we have to compare the results with a reference. Going forward, we will call "reference model" the run where we use a regular Peer Sampler on a 96-node 4-regular graph, where edges are assigned randomly. There are two reference models: one where data is iid and another where it is non-iid. We will be comparing the training/test losses and test accuracies of all the runs with those in Figure 4. It is worth mentioning that all runs use the same starting seed and parameters for data partitioning, so the same node will always have the same data across all runs.

6.1 Graph-building algorithm

Before showing the results however, the graph building algorithm by the TEE for this part is shown in Algorithm 2. It is imperative to build a 4-regular graph in the experiments, or else the results would not be comparable to the reference in Figure 4. The `top_neighbors` argument in the Bash script is the degree of the built graph, so it is 4 here.

Algorithm 2 Build a k-regular graph from a network of N nodes

```
Require: There are $N$ nodes. The TEE should have already filled the table S and sorted the
    similarities for each node.
 1: Input: an integer $k$                                    ▷ $k$ is the top_neighbors argument
 2: Output: a $k$-regular graph $G$ of $N$ vertices
 3: procedure isIncomplete($G$)                                                    ▷ $O(N)$
 4:     for $i \leftarrow 0$ to $N-1$ do
 5:         if length of $G[i] < k$ then return $True$
        return $False$
 6: procedure isDisconnected($G$)                                                 ▷ $O(Nk)$
 7:     Run BFS on $G$ starting from any node
 8:     if Set of visited nodes == Set of all nodes then return $False$ ▷ The graph is connected
        return $True$
 9: Initialize an empty dict $G$, mapping all node uids each to an empty set of neighbors.
10: $timeout \leftarrow k$
11: while isIncomplete($G$) AND $timeout > 0$ do
12:     $timeout \leftarrow timeout - 1$
13:     for $i \leftarrow 0$ to $N-1$ do
14:         if $G[i]$'s neighbors set has length $< k$ then
15:             for $peer \in S[i]$ do                         ▷ Remember that $S[i]$ is ordered
16:                 if $G[i]$'s neighbors set has length $< k$ then   ▷ Add edge between the nodes
17:                     Add $peer$ to $G[i]$
18:                     Add $i$ to $G[peer]$
19: if isIncomplete($G$) OR isDisconnected($G$) then
20:     Throw error
```

Analysis of Algorithm 2:

- **Complexity:** The double nested for-loops take $O(N^2)$ time at the worst case. However, it has been observed in practice that the bound is closer to $O(Nk)$ when $N \gg k$. And $timeout == k$, which means that the while-loop repeats at worst k times. Then, the total time complexity is $O(Nk^2)$

- **Completeness:** This algorithm is not complete, i.e., it will not always build a single-partition k-regular graph (see counterexample in Figure 5). However, as N grows, the likelihood of encountering failed cases decreases significantly, due to the randomness of data partitioning.

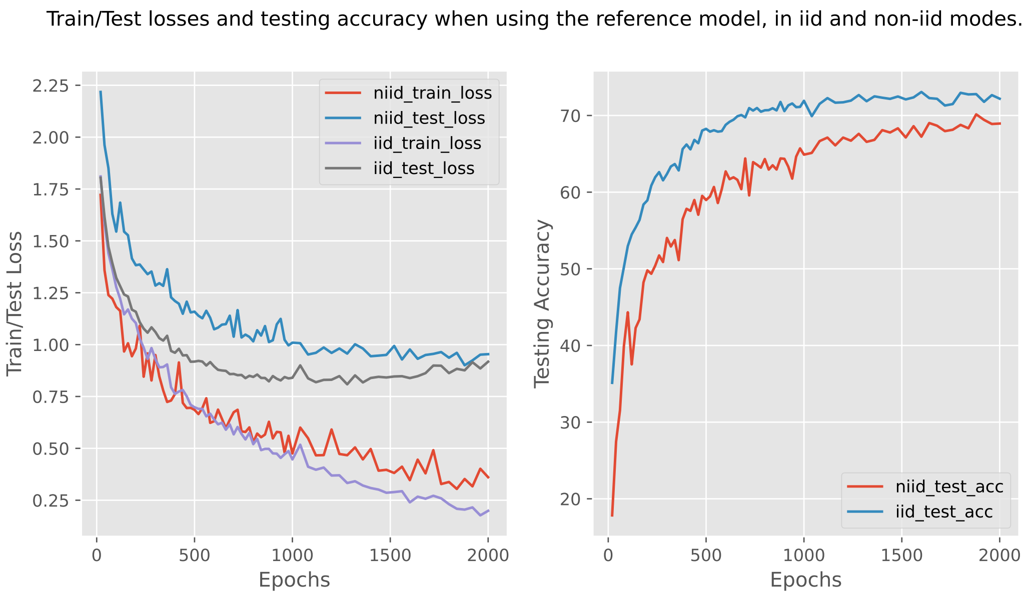

Figure 4: Initial results on a random 4-regular topology with iid data, without the similarity algorithm from the TEE

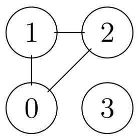

Figure 5: In this case, $N = 4, k = 2$. The following happened: 0 connected with 1, 1 with 2 and 2 with 0. It is now 3's turn, but there are no free nodes. The algorithm will timeout and throw an error, as 3 is disconnected from the rest.

6.2 Results

Note: In the following, what is meant by "least/most similar" model is that the TEE built a 4-regular topology by using a similarity table that was sorted from least similar to most similar node or vice versa.

6.2.1 Comparison among similarity metrics

In Table 1, I ranked all the major runs to compare the results with the references. There are three columns: train loss, test loss, and accuracy. They are each divided into a mean and a standard deviation. The values are derived from the last 200 iterations, in order to get a sense of the variance of the model. In my rankings, I took into account all the columns. For instance, a model that achieves good accuracy but slightly overfits the data (see row 2: reference model with iid data) might rank lower than another model that has lower accuracy but doesn't overfit as much (see row 1: Pearson correlation sorted by most negatively correlated nodes). We will analyse the results in 6.2.2.

6.2.2 Analysis of results

It is unsurprising to see that for the same metric, the models where the graph was created with the least similar nodes always perform better than those with the most similar ones: they overfit

		Train Loss		Test Loss		Accuracy (%)	
		Avg	Std	Avg	Std	Avg	Std
Pearson Correlation	Min correlation	0.545	0.057	0.849	0.011	71.568	0.319
Reference	iid	0.205	0.017	0.889	0.022	72.521	0.446
Cosine Similarity	Least similar	0.359	0.078	1.012	0.030	71.235	0.733
Reference	non-iid	0.345	0.034	0.937	0.022	69.088	0.628
Manhattan Distance	Least similar	0.388	0.067	1.111	0.024	69.610	0.501
Kullback-Leibler Divergence	Least similar	0.421	0.106	1.116	0.049	69.448	1.235
Euclidean Distance	Least similar	0.458	0.065	1.152	0.022	68.768	0.811
Pearson Correlation	Zero correlation	0.325	0.061	1.095	0.030	68.679	0.871
Cosine Similarity	Most similar	0.033	0.003	1.700	0.060	60.118	0.867
Euclidean Distance	Most similar	0.108	0.017	1.430	0.064	59.547	1.463
Pearson Correlation	Max correlation	0.100	0.012	1.457	0.023	57.281	0.682
Manhattan Distance	Most similar	0.099	0.007	1.565	0.050	55.106	0.948
Kullback-Leibler Divergence	Most similar	0.083	0.015	1.807	0.067	51.203	1.311

Table 1: Results of all the runs (ranked from best to worst) during the experiments. These numbers are an aggregation of the last 200 iterations for each run. *Note:* there are 3 runs for Pearson correlation because it is in $[-1, 1]$, so I took $\{-1, 0, 1\}$.

less the data and produce results with less variance and much better accuracy. We can clearly see this in Figure 6 (the metric used is cosine similarity):

- There is more than a 10% difference in accuracy at the end.
- The training loss for the model where nodes are sorted by most similar gets very close to 0; there is quasi-total overfitting. This might be due to nodes being coupled with others with similar data, so the local dataset used for training is not representative of the whole dataset as there are missing classes and not enough data homogeneity.

Now, let's get to the best performing models. Two metrics stand out by performing much better than the reference non-iid model, and by performing as well, if not better than the iid case:

1. Pearson correlation when connecting nodes with negatively correlated label distributions.
2. Cosine similarity when connecting nodes with orthogonal label distributions.

We can see the results of these metrics and the random graph iid case in Figure 7. Here are the main points that we can extract from this figure:

- All three models converge to an accuracy of over 71%.
- Cosine similarity is the metric that mimics best the iid case: the curves for their respective evaluation functions have similar shapes. Both models also start overfitting at later iterations (we can see the green and gray curves on the left graph lean upwards at the end).
- This overfitting is not present when using Pearson correlation: even though the model doesn't start as well as the other two, the test loss function (yellow) never stops going down and eventually reaches lower values than the other two functions. The train loss function (blue) also fits better the overall dataset.

One question arises from these observations: **why do iid models start overfitting at some point whereas well designed non-iid models don't?**

This could be due to the fact that if a model is trained on iid data and then tested on the same distribution of data, it may overfit because it has seen all the data and learned patterns that may not generalize to new, unseen data. In other words, the model may perform well on the training data but poorly on the testset, because it has learned to "memorize" the training data rather than learn more general patterns. On the other hand, if a model is trained on non-iid data, it may not overfit as easily because the data distribution is more diverse and there may be less opportunity for the model to "memorize" specific patterns in the data. As a result, the model may be more robust and perform better on unseen data. It's worth noting that overfitting can still occur in the non-iid case when topologies are not built properly, as we see with the lower ranked models in Table 1.

6.2.3 Conclusion of the experiments

The main question behind this project was: **Can we use the label distribution of a node to assign to it appropriate neighbors and reduce the negative effects of non-iidness?**

These experiments showed that we can reduce the negative effects of non-iidness by comparing label distributions of nodes pairwise using cosine similarity. They also showed that we can reduce the negative effects of iidness as well (overfitting): using Pearson correlation between label distributions and building topologies by connecting negatively correlated nodes solves both issues.

This means that Pearson correlation is the way to go. However, for a low number of iterations, one should use cosine similarity due to its faster convergence.

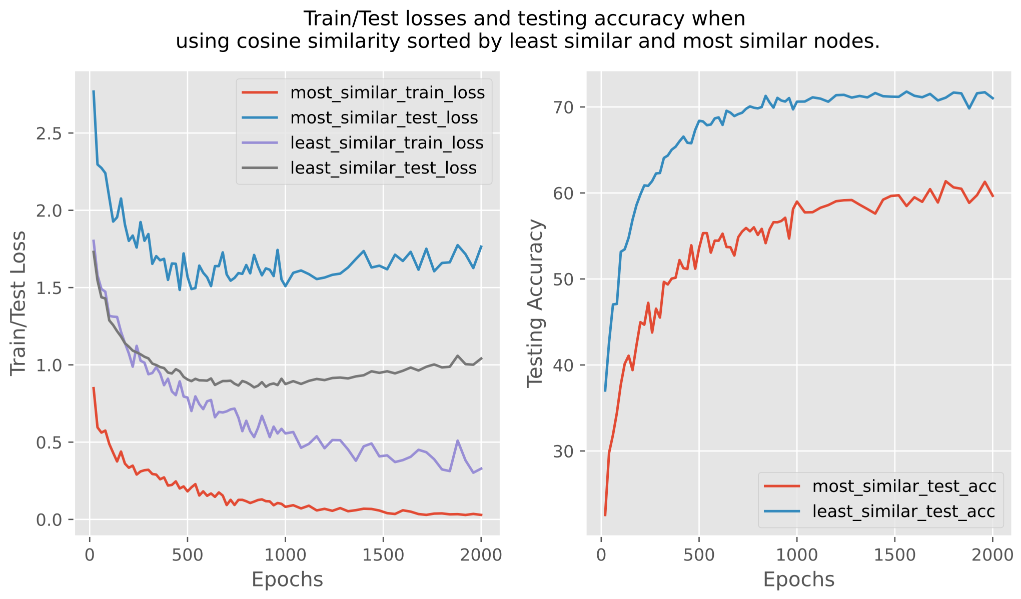

Figure 6: Train/test losses and testing accuracies with cosine similarity used in two cases: when we sort the similarities ascending and descending.

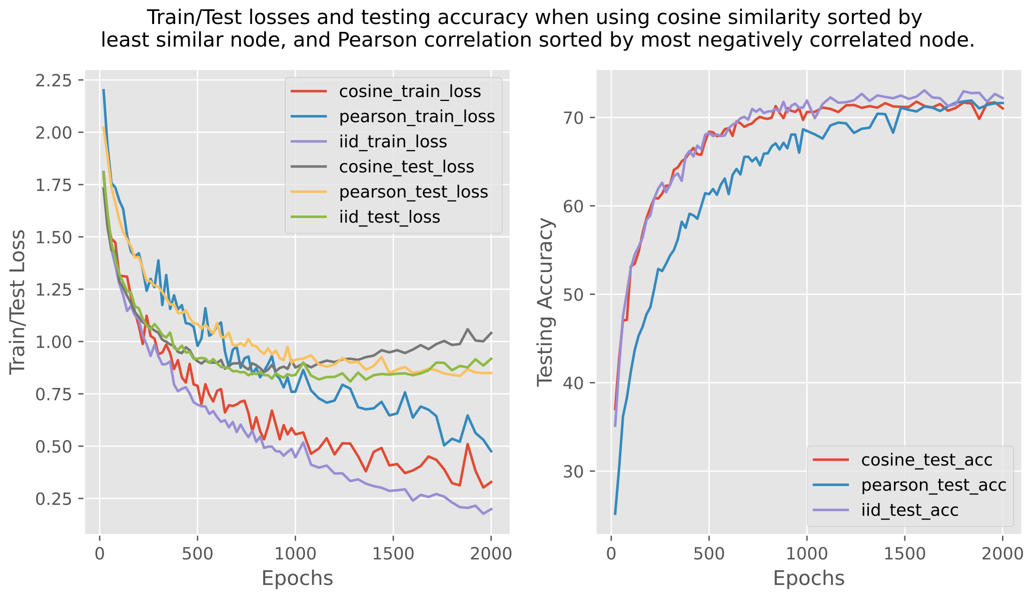

Figure 7: Train/test losses and testing accuracies for three cases: the reference run with iid data, a run where we use cosine similarity and sort by least similar nodes, and a run where we use Pearson correlation and sort by most negatively correlated nodes.

7 New Algorithms

As we saw in the last section, using cosine similarity or Pearson correlation is very effective in reducing data heterogeneity and producing results similar to iid when running in non-iid mode. Hence in all algorithms presented in this section that use a TEE, I will present the results for both similarities.

Note: In the framework, the model does not converge if the topology used is not connected, i.e., there exists a path from any node to any other node.

7.1 Use cases for similarity metrics

We can see in Figure 8 the results of two extreme cases with a regular Peer Sampler:

1. When using a fully connected (FC) topology, each node communicates with all other nodes, which is why we reach the best possible results with this dataset and these parameters, no matter the iidness of the data. For the last 200 iterations, test loss is as low as 0.754 with a standard deviation of just 0.001. Also, the test accuracy is at 74%.

2. A ring is the smallest connected topology where each node has at least 2 neighbors. At his the highest bias and highest variance of any other model.

This means that in theory, no model can reach better results than FC and worse results than a ring (other than a star and a line, but we don't consider those: they have nodes with degree < 2). However, we cannot always use a FC topology, because there are communication costs that grow linearly with the number of neighbors per node: In a FC 96-node framework, every node sends $34.152MB$ of data per iteration, whereas in a 4-regular framework, each node send $1.438MB$ of data per iteration (23.75 times less).

7.1.1 Reduce communication costs

We will propose here some algorithms that and try to reach results close to a FC topology, without as much communication costs. We are always working with non-iid data.

Top Neighbors Pairing This algorithm is very simple: After computing the similarities, there is a connection between two nodes if:

- **Cosine similarity:** their label distributions are orthogonal, i.e., the angle between them $\theta = \frac{\pi}{2} \implies \cos(\theta) = 0$.

- **Pearson correlation:** their label distributions are negatively correlated enough (here, I defined "enough" as smaller than -0.4)

After several runs with 96 nodes and non-iid data from CIFAR-10, I noticed the following statistics regarding the label distributions:

- Each node has on average 13.9 orthogonal nodes.

- Each node has on average 10.0 nodes with Pearson correlation < -0.4

Some nodes don't have good neighbor candidates, so we also add a ring around the topology to ensure that the resulting graph is connected, i.e., all nodes belong to the same partition. This brings the number of neighbors per node to 15.64 for cosine similarity and 11.875 for Pearson correlation. Thus, the algorithm reduces the data sent by a node each iteration by a factor of 6 and 8 respectively. The results are shown in Figure 9. A few observations from this figure: firstly, the test losses and accuracies have low bias and low variance. Secondly, the differences with a fully connected topology are close, so if one's priority is saving on communication costs, the Top Neighbors Pairing algorithm could be a good tradeoff. Finally, Pearson correlation clearly outperforms cosine similarity in this algorithm (more than 1% difference in accuracy), especially considering that it creates 4 less neighbors per node, so more data is saved as well.

7.1.2 Preserve Privacy

In the case where nodes don't want to share their label distributions even to the enclave, but only want to say which classes they own in their local training dataset, the TEE can assign to each node a vector D, such that for every class i, $D_{node}[i] = 1 \iff i \in$ local training data of the node, and 0 otherwise. Now the TEE can consider these vectors as the label distributions for the nodes and use cosine similarity to decide how to pair nodes together.

7.1.3 Create clusters of nodes

There is one issue with building k-regular graphs when doing pairwise comparisons of nodes: if node a is dissimilar to both nodes b and c, the latter nodes can be similar to each other, which means that a learns the same from b or c, which is a waste of a neighbor. Algorithm 3 shows a way to group nodes into fully connected communities instead of pairs.

Even though it does not use similarity metrics to compare nodes, the algorithm is inspired from cosine similarity, and more precisely the fact that if two nodes are orthogonal, the intersection of the sets of classes in their respective local data is empty. This is how this greedy algorithm works: we add to a cluster the node that has the least in common with the union of classes of that cluster (hence the name max-contribution clustering).

Algorithm 3 Max-Contribution Clustering

```
Require: There are $N$ nodes. The TEE should have already created the set of classes possessed
    by each node.
 1: Input: an integer $k$                                   ▷ $k$ is the top_neighbors argument
 2: Output: a graph $G$ with $\frac{N}{k}$ fully connected clusters of $k$ vertices each
 3:
 4: procedure GETMAXCONTRIBUTION($n$, $c_i$, $l_i$)                                  ▷ $O(N)$
 5:     $maxContribution \leftarrow 0$
 6:     $winnerNode \leftarrow NULL$
 7:     for $j \leftarrow 0$ to $N-1$ do
 8:         if $j \notin c_i$ then
 9:             $diff \leftarrow classes(j) \setminus l_i$                 ▷ This is set difference
10:             if $size(diff) > maxContribution$ then
11:                 $winnerNode \leftarrow j$
12:                 $maxContribution \leftarrow size(diff)$
        return $winnerNode$                  ▷ Choose the node that adds the most classes to $l_i$
13: Initialize an empty dict $G$, mapping all node uids each to an empty set of neighbors.
14: Initialize a set $n$ containing all node IDs from 0 to $N-1$
15: for $i \leftarrow 0$ to $k-1$ do
16:     initialize empty set $c_i$                                 ▷ $c_i$ is a set of nodes
17:     initialize empty set $l_i$                                 ▷ $l_i$ is a set of classes
18:     while $size(c_i) < k$ and $size(n) > 0$ do     ▷ Make sure that there still are nodes to add
19:         $winner \leftarrow$ GETMAXCONTRIBUTION($n, c_i, l_i$)
20:         Add $winner$ to $c_i$
21:         $l_i \leftarrow l_i \cup classes(winner)$
22:     For each node in $c_i$ add in $G[i]$ all other nodes in the cluster.
```

Note: At the time of writing, this algorithm has not been implemented in the code.
Analysis of Algorithm 3:

- **Complexity:** the *getMaxContribution* method has complexity $O(N)$, and it is called once for every node. So in total, the algorithm runs in $O(N^2)$

- **Connectivity:** To make sure that the resulting graph is connected, choose a cluster representative from each cluster and form a ring between them. The resulting topology looks like Figure 10.

- **Neighbors per node:** There are $\frac{N}{k}$ clusters of size k each (if k does not divide N, the last cluster will be smaller, but for simplicity's sake, let k divide N), so $k-1$ neighbors per node. Each cluster representative has 2 more neighbors to form the ring. The total neighbors per node is $\frac{N(k-1)+2k}{N} = k - 1 + \frac{2k}{N}$, which is roughly $k-1$ when $N \gg k$

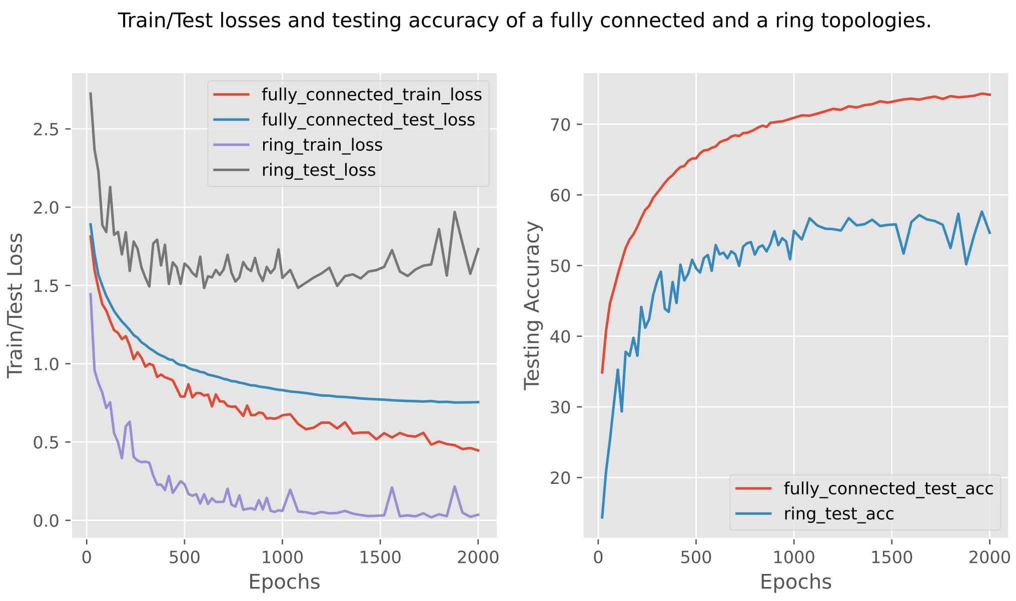

Figure 8: Results of the two extreme topologies: a fully connected graph and a ring

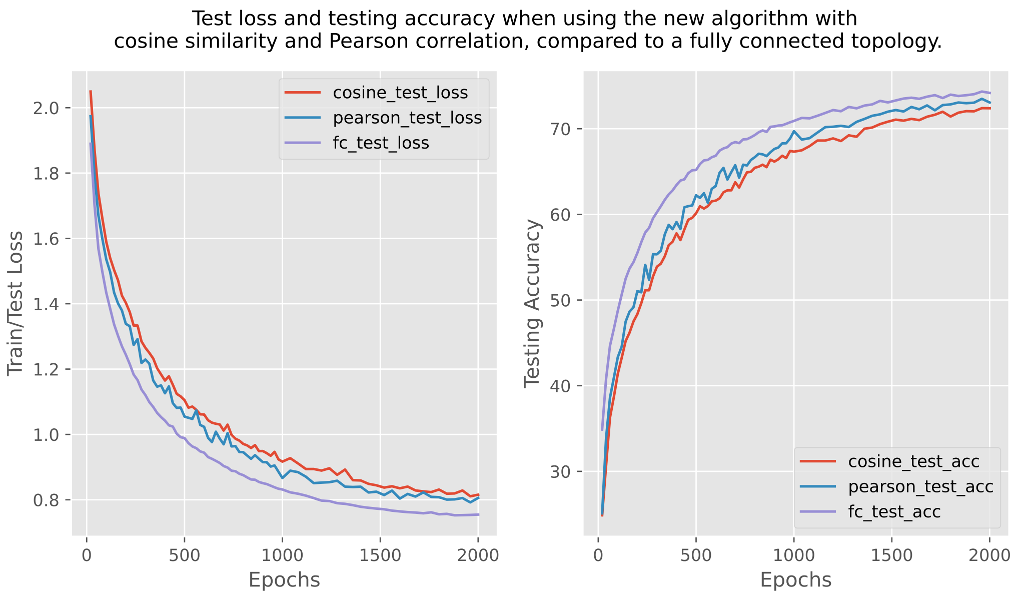

Figure 9: Results for the Top Neighbors Pairing algorithm with cosine similarity and Pearson correlation, compared to a fully connected topology.

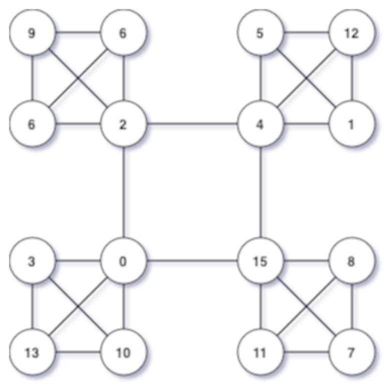

Figure 10: Topology of Max-Contribution Clustering for $N = 16$ and $k = 4$

8 Conclusion

In this project, the aim was to minimize the negative effects of data heterogeneity by building topologies that use the label distribution of nodes. We defined and analysed different similarity metrics to compare the label distribution of two nodes. We also created a C interface acting as an oracle that answers queries about the similarity of two label distributions. This oracle can be stored securely in a TEE for security and privacy reasons.

The first observation is that nodes with dissimilar distributions should be connected together in order to create clusters that represent better the global distribution of the data. Secondly, some similarity metrics are more successful than others in quantifying the dissimilarity that we are looking for. More precisely, it was found that cosine similarity and Pearson correlation are very effective metrics, and we were able to produce better results than some reference topologies, thus proving the effectiveness of these metrics.

Finally, this project proposed some use cases for the similarity metrics. They can be used to build more efficient topologies that can produce very good results with much less data traffic than other random topologies. They can also be used in more privacy-preserving systems. Lastly, we proposed a way to cluster nodes in a framework by neighborhoods of many nodes instead of just pairwise connections.

8.1 Further Works

As future work, one could do the following:

- Expand the experiments on datasets other than CIFAR-10 to see if the results generalize or not.
- Try to find new similarity metrics that can be more effective than cosine similarity or Pearson correlation.
- Do the same experiments but with dynamic topologies.
- Build an actual enclave and look at the performance overhead of the built TEE.

References

[1] Alan Bundy and Lincoln Wallen. “Breadth-First Search”. In: *Catalogue of Artificial Intelligence Tools*. Ed. by Alan Bundy and Lincoln Wallen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 13–13. ISBN: 978-3-642-96868-6. DOI: `10.1007/978-3-642-96868-6_25`. URL: `https://doi.org/10.1007/978-3-642-96868-6_25`.

[2] Sebastian Caldas et al. “Leaf: A benchmark for federated settings”. In: *arXiv preprint arXiv:1812.01097* (2018).

[3] Xuhui Chen et al. “When machine learning meets blockchain: A decentralized, privacy-preserving and secure design”. In: *2018 IEEE international conference on big data (big data)*. IEEE. 2018, pp. 1178–1187.

[4] *DecentralizePy*. SaCS. Nov. 9, 2021. URL: `https://gitlab.epfl.ch/sacs/decentralizepy` (visited on 06/17/2022).

[5] Akash Dhasade et al. “TEE-based decentralized recommender systems: The raw data sharing redemption”. In: *arXiv preprint arXiv:2202.11655* (2022).

[6] Pieter Hintjens. *ZeroMQ: messaging for many applications.* " O’Reilly Media, Inc.", 2013.

[7] Kevin Hsieh et al. “The Non-IID Data Quagmire of Decentralized Machine Learning”. In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 4387–4398. URL: `https://proceedings.mlr.press/v119/hsieh20a.html`.

[8] *Joseph Abboud Semester Project*. SaCS. Nov. 9, 2021. URL: `https://gitlab.epfl.ch/abboud/decentralizepy/-/tree/label-distribution` (visited on 06/17/2022).

[9] Alex Krizhevsky, Geoffrey Hinton, et al. “Learning multiple layers of features from tiny images”. In: (2009).

[10] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), pp. 436–444.

[11] Yann LeCun et al. “Backpropagation applied to handwritten zip code recognition”. In: *Neural computation* 1.4 (1989), pp. 541–551.

[12] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[13] Xiangru Lian et al. “Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent”. In: *Advances in Neural Information Processing Systems* 30 (2017).

[14] Jukka Määttä, Abdenour Hadid, and Matti Pietikäinen. “Face spoofing detection from single images using texture and local shape analysis”. In: *IET biometrics* 1.1 (2012), pp. 3–10.

[15] Brendan McMahan et al. “Communication-efficient learning of deep networks from decentralized data”. In: *Artificial intelligence and statistics*. PMLR. 2017, pp. 1273–1282.

[16] Univeristy of Toronto. *CIFAR-10 Dataset*. URL: `https://www.cs.toronto.edu/~kriz/cifar.html`.

[17] Stefanie Warnat-Herresthal et al. “Swarm learning for decentralized and confidential clinical machine learning”. In: *Nature* 594.7862 (2021), pp. 265–270.

[18] Jeffrey Wigger. “Improving communication-efficiency in Decentralized and Federated Learning Systems”. MA thesis. Ecole Polytechnique Fédérale de Lausanne, 2021.