UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Reduced Communication Purely Distributed Federated Learning

Diploma Thesis

Koutsoukis Nikolaos

Supervisor: Dimitrios Katsaros

January 2024

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Reduced Communication Purely Distributed Federated Learning

Diploma Thesis

Koutsoukis Nikolaos

Supervisor: Dimitrios Katsaros

January 2024

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Πλήρως Κατανεμημένη Ομόσπονδη Μάθηση Ελαττωμένης Επικοινωνίας

Διπλωματική Εργασία

Κουτσούκης Νικόλαος

Επιβλέπων: Δημήτριος Κατσαρός

Ιανουάριος 2024

Approved by the Examination Committee:

Supervisor **Dimitrios Katsaros**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Dimitrios Rafailidis**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Georgios Thanos**
Laboratory Teaching Staff, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

I want to thank my supervisor, Associate Professor Dimitrios Katsaros, for his guidance and the time he dedicated to providing valuable advice and instructions for completing my thesis. His support has been instrumental in shaping the outcome of my work. Also, I'd like to thank Associate Professor Dimitrios Rafailidis and Laboratory Teaching Staff Georgios Thanos for their contributions to my thesis review committee.

Additionally, I extend my heartfelt thanks to my parents, who have consistently supported me throughout my academic journey. I owe the entirety of my educational path to them, as they have played a crucial role in my studies up to this point. Finally, I would like to thank my friends, especially Alexandros Stoltidis, for his civil help, as he offered me his computer to calculate the final experiments.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Koutsoukis Nikolaos

Diploma Thesis

Reduced Communication Purely Distributed Federated Learning

Koutsoukis Nikolaos

Abstract

Nowadays, the proliferation of Internet of Things (IoT) devices and cloud-based applications across isolated data centers has sparked considerable interest in refining models to address the privacy and overhead challenges inherent in centralized machine learning. In response to these challenges, federated learning has emerged as a viable solution, allowing multiple devices connected via wireless networks to train machine learning models collaboratively without exposing raw data.

Nearly all prior works focus on star topologies, featuring distributed clients or nodes and a central server. However, this approach raises concerns about robustness and demands substantial communication resources. To address the challenges, we propose a fully decentralized federated deep learning model customized for wireless networks. Our design involves a clustered network that includes a cluster head node. This local server is an intermediary between the cluster nodes and facilitates averaging implementation.

A notable innovation in our model is incorporating a node participation criterion based on the highest accuracy achieved by each node. This strategic approach significantly reduces communication costs between nodes. To validate the effectiveness of our model, we conducted a diverse set of experiments using real-world datasets. Our implementation involved leveraging various tools and platforms to ensure a comprehensive evaluation of our analytical results and showcase our proposed model's efficiency.

Keywords:

Federated Learning, wireless networks, fully distributed, node participation, ML model, clustering

Διπλωματική Εργασία

Πλήρως Κατανεμημένη Ομόσπονδη Μάθηση Ελαττωμένης Επικοινωνίας

Κουτσούκης Νικόλαος

Περίληψη

Στις μέρες μας, ο πολλαπλασιασμός των συσκευών του Διαδικτύου των Πραγμάτων (IoT) και των εφαρμογών που βασίζονται στο cloud σε απομονωμένα κέντρα δεδομένων έχει προκαλέσει σημαντικό ενδιαφέρον για την τελειοποίηση των μοντέλων για την αντιμετώπιση των προκλήσεων της ιδιωτικής ζωής και του υπερβολικού κόστους που συνεπάγεται η κεντρική μηχανική μάθηση. Ως απάντηση σε αυτές τις προκλήσεις, η ομοσπονδιακή μάθηση έχει αναδειχθεί ως μια βιώσιμη λύση, επιτρέποντας σε πολλαπλές συσκευές που συνδέονται μέσω ασύρματων δικτύων να εκπαιδεύουν συνεργατικά μοντέλα μηχανικής μάθησης χωρίς να εκθέτουν τα δεδομένα τους.

Σχεδόν όλες οι προηγούμενες εργασίες επικεντρώνονται σε τοπολογίες αστέρα, με κατανεμημένους πελάτες ή κόμβους και έναν κεντρικό διακομιστή. Ωστόσο, αυτή η προσέγγιση εγείρει ανησυχίες σχετικά με την αξιοπιστία και απαιτεί σημαντικούς πόρους επικοινωνίας. Προκειμένου να αμβλυνθούν αυτές οι προκλήσεις, η πρότασή μας εισάγει ένα πλήρως αποκεντρωμένο ομοσπονδιακό μοντέλο βαθιάς μάθησης προσαρμοσμένο για ασύρματα δίκτυα. Σχεδιάζουμε συγκεκριμένα ένα δίκτυο σε συστάδες, ενσωματώνοντας έναν κόμβο επικεφαλής συστάδας που χρησιμεύει ως τοπικός διακομιστής μεταξύ των κόμβων συστάδας, διευκολύνοντας την εφαρμογή της μέσης μάθησης.

Μια αξιοσημείωτη καινοτομία στο μοντέλο μας είναι η ενσωμάτωση ενός κριτηρίου συμμετοχής κόμβων με βάση την υψηλότερη ακρίβεια που επιτυγχάνει κάθε κόμβος. Αυτή η στρατηγική προσέγγιση μειώνει σημαντικά το κόστος επικοινωνίας μεταξύ των κόμβων. Για να επικυρώσουμε την αποτελεσματικότητα του μοντέλου μας, πραγματοποιήσαμε μια σειρά ποικίλων πειραμάτων χρησιμοποιώντας σύνολα δεδομένων του πραγματικού κόσμου. Η υλοποίησή μας περιελάμβανε την αξιοποίηση διαφόρων εργαλείων και πλατφορμών για να διασφαλίσουμε μια ολοκληρωμένη αξιολόγηση των αναλυτικών μας αποτελεσμάτων και να αναδείξουμε την αποτελεσματικότητα του προτεινόμενου μοντέλου μας.

Λέξεις-κλειδιά:

Ομοσπονδιακή Μάθηση, ασυρματα δίκτυα, πλήρως κατανεμημενο, συμμετοχή κόμβων, μοντέλο ML, ομαδοποίηση

Table of contents

List of figures

Abbreviations

IoT	Internet of Things
ML	Machine Learning
AI	Artificial Intelligence
FL	Federated Learning
ANNs	Artificial Neural Networks
DNNs	deep neural networks
CNNs	Convolutional Neural Networks
RNNs	Recurrent Neural Networks
DBNs	Deep Belief Networks
ReLU	Rectidied Linear Unit
PDFL	Purely Distributed Federated Learning
ANP	All Node Participation
RNP	Random Node Participation
BANP	Best Accurancy Node Participation
FDA	Full Docker Architecture
HDA	Hybrid Docker Architecture

Chapter 1

Introduction

Nowadays, has seen a surge in the prevalence of Internet of Things (IoT) devices, catering to diverse applications such as mobile phones, smart healthcare, wearables, autonomous vehicles, and augmented reality. The increasing computational capabilities of these devices, coupled with concerns about data privacy, have given rise to technologies that maintain datasets locally on the devices, emphasizing edge network computation.

The approach to model training involved a central node, typically a server, in what is known as centralized machine learning (ML). However, as the volume of data generated by these devices grows, along with increased capacity and energy consumption, there is a rationale for optimizing local resources. Moreover, privacy-sensitive applications often entail reluctance to transmit raw data from end users. In response to these considerations, Federated Learning (FL) has emerged as a methodology for distributed model training across devices.

Traditional federated learning (FL) [1] generally consists of a central node (server) linked to numerous devices (clients). In each cycle of model training, like a neural network, two essential steps are involved. First, local training takes place, where each device updates its local model using its dataset and the global model through gradient descent methods. Next, global aggregation occurs, where the main server collects all local updates, calculates a new global model, and then synchronizes all nodes to commence the subsequent round of training with this aggregated version.

A significant benefit of federated learning lies in its ability to address privacy concerns by ensuring that raw data is never exposed between clients and the main server. In conventional federated learning, a server is essential for facilitating communication and aggregating parameters. However, this approach proves limiting and impractical in contemporary networks,

where devices are expected to operate independently without a central server and communicate directly with adjacent ones.

This thesis proposes a fully decentralized federated deep learning model within clustered wireless networks to overcome communication and computation constraints. In essence, our network is partitioned into clusters, and we designate the node with the highest closeness centrality as the clusterhead. This node assumes the role of the local server among the cluster nodes and is responsible for local averaging. Additionally, we incorporate global averaging, involving data exchange between different clusters. Our focus extends to reducing communication costs and energy consumption associated with data transmission.

To accomplish these goals, we implement a best-accuracy approach for node participation in averaging. In this method, nodes with the highest accuracy in their neural networks are chosen to partake in training and share their data with the clusterhead (local server). Our model's evaluation involved three distinct approaches in our experiments: complete participation of all nodes in the network, random participation based on a selected threshold, and participation based on best accuracy.

1.1 Thesis Structure

This thesis is structured with a comprehensive approach:

In Chapter 2, we provide an extended introduction to key concepts such as Deep and Federated Learning. Additionally, we explore clustering algorithms, centrality algorithms, and review related work in the field of Federated Learning.

Moving on to Chapter 3, we present our novel model for federated learning in wireless networks. Here, we detail the dataset used in our model, discuss innovations in the federation process, and elaborate on the algorithms employed to achieve our goals.

Chapter 4 focuses on the implementation of the project. We delve into the tools, platforms and programming libraries used during its creation. This chapter provides a detailed description of how these resources were used during the construction of the project. Finally, we conclude by explaining the functionality of the project.

In Chapter 5, we delve into the experiments conducted and analyze the results obtained.

Finally, Chapter 6 serves as a conclusion, summarizing our work and initiating discussions on potential directions for future research.

Chapter 2

Background

2.1 Deep Learning

Deep learning [2], a subset of machine learning, leverages artificial neural networks to uncover intricate patterns and relationships within data. Unlike traditional programming, deep learning does not necessitate explicit instructions for every task. Its popularity has surged in recent years, driven by advancements in processing power and the abundance of extensive datasets. Rooted in artificial neural networks (ANNs), also known as deep neural networks (DNNs), these structures draw inspiration from the biological neurons in the human brain, designed to glean insights from substantial datasets.

Deep Learning, a branch within the realm of Machine Learning, employs neural networks as a foundational framework for modeling and address complex problems. These networks mirror the structure and function of the human brain, comprising interconnected nodes organized in layers to process and transform data.

A defining feature of Deep Learning is the deployment of deep neural networks, characterized by multiple layers of interconnected nodes. These networks excel at learning complex representations by identifying hierarchical patterns and features in the data, eliminating the need for manual feature engineering.

Deep Learning has witnessed remarkable success in diverse domains, including image recognition, natural language processing, speech recognition, and recommendation systems. Notable architectures include Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Deep Belief Networks (DBNs).

Training deep neural networks typically demands substantial data and computational re-

sources. However, the advent of cloud computing and specialized hardware, such as Graphics Processing Units (GPUs), has eased the training process for deep neural networks.

In summary, Deep Learning, a subfield of Machine Learning, utilizes deep neural networks to model and address complex problems. Its significant success across various domains is expected to persist as data availability increases, and more powerful computing resources become accessible.

So, Deep learning is the branch of machine learning which is based on artificial neural network architecture.An artificial neural network (ANN) employs layers of interconnected nodes known as neurons to collaboratively process and gain insights from input data. In a fully connected deep neural network, an input layer is followed by one or more hidden layers linked sequentially. Each neuron receives input from the preceding layer's neurons or the input layer. The output from one neuron serves as input for other neurons in the subsequent layer, and this iterative process continues until the final layer generates the network's output. These layers undertake nonlinear transformations on the input data, enabling the network to acquire intricate representations of the input data.

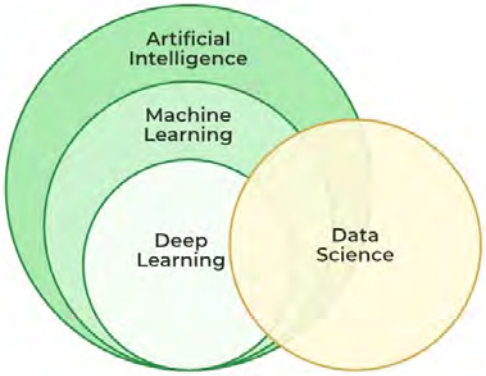


Figure 2.1: Relationship between AI, ML and DL

In contemporary times, deep learning has emerged as one of the most widely embraced and prominent domains within machine learning. Its widespread adoption is attributable to its remarkable success in diverse applications, including computer vision, natural language processing, and reinforcement learning [3].

Deep learning finds applicability across various machine learning paradigms, encompassing supervised, unsupervised, and reinforcement learning. Its utilization involves diverse methods for processing data [4].

- **Supervised Machine Learning:** Supervised machine learning involves the neural net-

work learning to make predictions or classify data based on labeled datasets. Inputting both input features and target variables, the neural network refines its predictions through backpropagation, minimizing the difference between predicted and actual targets. Deep learning algorithms, including Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), excel in supervised tasks such as image classification, recognition, sentiment analysis, and language translation.

- **Unsupervised Machine Learning:** Unsupervised machine learning sees the neural network learning to identify patterns or cluster datasets based on unlabeled datasets. Absent target variables, the machine autonomously determines hidden patterns or relationships within the datasets. Deep learning algorithms like autoencoders and generative models shine in unsupervised tasks like clustering, dimensionality reduction, and anomaly detection.

- **Reinforcement Machine Learning:** Reinforcement Machine Learning involves an agent learning to make decisions in an environment to maximize a reward signal. The agent interacts with the environment, taking actions and observing resulting rewards. Deep learning is harnessed to learn policies, or sets of actions, optimizing cumulative rewards over time. Reinforcement learning algorithms like Deep Q Networks and Deep Deterministic Policy Gradient (DDPG) are applied to reinforce tasks such as robotics and game playing.

2.1.1 Artificial neural networks

Artificial Neural Networks consist of artificial neurons, referred to as units, organized in layers that collectively form the entire neural network system. The number of units in a layer can vary, ranging from a few dozen to millions, depending on the complexity required for the network to uncover hidden patterns in the dataset. Typically, an Artificial Neural Network comprises an input layer, an output layer, and hidden layers. The input layer receives external data for analysis or learning by the neural network. Subsequently, this data undergoes transformation in one or more hidden layers, producing information valuable for the output layer. Ultimately, the output layer generates a response representing the neural network's output in response to the provided input data.

In most neural networks, units are interconnected between layers, and each connection

is assigned weights that determine the influence of one unit on another. As data traverses from one unit to another, the neural network progressively learns more about the dataset, culminating in an output from the output layer.

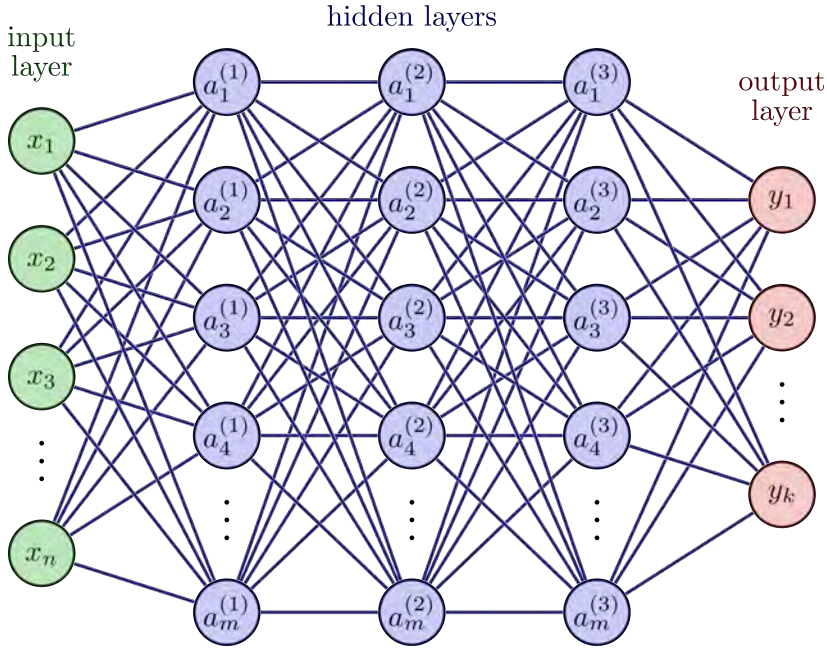


Figure 2.2: Deep Neural Network

The structures and functions of human neurons form the foundation for artificial neural networks, also known as neural networks or neural nets. In an artificial neural network, the initial layer is the input layer, serving as the first stage. This layer acquires input from external sources and transmits it to the subsequent layer, known as the hidden layer.

Within the hidden layer, each neuron takes input from neurons in the preceding layer, computes the weighted sum, and transmits the result to the neurons in the subsequent layer. These connections involve weighted values, where the impact of inputs from the previous layer is fine-tuned by assigning distinct weights to each input. During the training process, these weights are adjusted to optimize model performance, ensuring an enhanced ability to learn and make accurate predictions.

2.1.2 Types of neural networks

Neural networks exhibit various types [5], each designed for specific purposes. While this overview doesn't encompass all possible classifications, it provides a representative selection of common neural network types frequently encountered in practical applications:

- **Feedforward Neural Network (FNN):** The FNN, a fundamental artificial neural network, directs data or input in a single direction. Input enters through the input layer, traverses the network without hidden layers (or with optional hidden layers), and exits through the output layer. Typically, the FNN follows a front-propagated wave without incorporating backpropagation.

- **Convolutional Neural Network (CNN):** Similar to the feedforward neural network, the CNN involves weighted connections determining the influence of one unit on another. Distinguishingly, a CNN incorporates one or more convolutional layers, utilizing convolution operations on input data and forwarding the results to subsequent layers. Particularly valuable in computer vision, CNNs find applications in speech and image processing.

- **Modular Neural Network:** Comprising distinct neural networks operating independently, the Modular Neural Network focuses on obtaining outputs without inter-network interaction. Each network handles a specific sub-task, utilizing unique inputs compared to other networks. The modular approach simplifies complex computational processes by breaking them down into smaller, manageable components.

- **Radial Basis Function Neural Network (RBFNN):** RBF functions, which consider the distance of a point concerning the center, characterize RBFNNs. Consisting of two layers, the first layer maps input to all radial basis functions in the hidden layer. RBFNNs are adept at modeling data that represents underlying trends or functions.

- **Recurrent Neural Network (RNN):** RNNs retain the output of a layer and feed it back to the input for enhanced outcome prediction. Similar to feedforward neural networks, RNNs initiate after computing the first layer's output. Each unit in subsequent layers retains information from the previous step, functioning as a memory cell for computations.

2.1.3 Activation functions

Activation functions [6] play a crucial role in neural networks by calculating the weighted sum of inputs and biases. This computation determines whether a neuron should be activated. The activation function manipulates the input data, generating an output for the neural net-

work that encompasses the data's parameters. In certain literature, these functions are alternatively known as transfer functions. They exhibit flexibility, being categorized as either linear or nonlinear based on the specific function they represent. These functions are instrumental in governing the output of neural networks and find application across various domains.

Types of Activation Functions

The different kinds of activation functions include:

- **Linear Activation Function:** Also known as a straight-line function, the linear activation function ensures that the activation is directly proportional to the input, i.e., the weighted sum from neurons. It is represented by this equation:

$$f(x) = ax + c$$

  However, a drawback of this activation function is its inability to be confined to a specific range. When applied to all nodes, it transforms the activation function into a linear regression, causing the final layer of the Neural Network to function as a linear transformation of the first layer. Another issue arises during gradient descent, where differentiation yields a constant output. This constant rate of change in error can adversely impact the output and disrupt the logic of backpropagation.

- **Non-Linear Activation Function** are known to be the most used activation functions. It allows the adaptation of a neural network model to a wide range of data and the differentiation of results.

  The range or curves of these functions serve as the primary basis for division:

  - **Sigmoid Activation Function** takes a real value as the input and outputs another value between 0 and 1. The sigmoid activation function translates the input ranged in (-∞,∞) to the range in (0,1)
  - The **Tanh Activation Function** is just another possible function that can be used as a non-linear activation function between layers of a neural network. It shares a few things in common with the sigmoid activation function. Unlike a sigmoid function that will map input values between 0 and 1, the Tanh will map values between -1 and 1. Similar to the sigmoid function, one of the interesting properties

of the tanh function is that the derivative of tanh can be expressed in terms of the function itself.

- **ReLU Activation Functions**: The formula is deceptively simple: max(0,z). Despite its name, Rectified Linear Units, it's not linear and provides the same benefits as Sigmoid but with better performance.

  Leaky Relu is a variant of ReLU. Instead of being 0 when $z<0$, a leaky ReLU allows a small, non-zero, constant gradient α (normally, $\alpha=0.01$). However, the consistency of the benefit across tasks is presently unclear. Leaky ReLUs attempt to fix the "dying ReLU" problem.

  Parametric Relu gives the neurons the ability to choose what slope is best in the negative region. They can become ReLU or leaky ReLU with certain values of α.

- **Maxout activation:**. The Maxout activation serves as an extension of the ReLU and leaky ReLU functions, constituting a piecewise linear function that outputs the maximum of inputs. Specifically designed to complement the dropout regularization technique, both ReLU and leaky ReLU can be regarded as special cases of Maxout. Notably, the Maxout neuron inherits the advantages of a ReLU unit without encountering the drawbacks associated with the phenomenon of dying ReLU. However, it necessitates a doubling of the total number of parameters for each neuron, thereby requiring a higher number of parameters for training.

- **Exponential Linear Unit (ELU)**: it stands out as a function that exhibits faster convergence and yields more accurate results. Distinguishing itself from other activation functions, ELU introduces an additional alpha constant, a positive numerical value. ELU shares similarities with ReLU for non-negative inputs, both adhering to the identity function form. Conversely, ELU undergoes a gradual smoothing process for negative inputs, extending until its output reaches $-\alpha$, whereas ReLU undergoes a sharp smoothing.

- **Softmax** Activation Function performs calculations to determine the probability distribution of an event across 'n' different events. In a broad sense, this function computes the probabilities associated with each target class among all conceivable target classes. Subsequently, the derived probabilities play a crucial role in ascertaining the target class for a given set of inputs.

2.1.4 Loss functions

The primary objective of a deep learning neural network is to establish a mapping from a given set of inputs to a corresponding set of outputs. However, achieving perfection in this process is inherently challenging. To gauge the accuracy of the training and evaluate how effectively the algorithm aligns with the data, loss functions come into play [7]. In neural networks, the overarching goal is typically to minimize the model's error during the optimization process, necessitating the selection of an optimal loss/cost function.

Loss functions fall into two principal categories, contingent upon the nature of our learning objectives: classification and regression losses. In classification, the endeavor is to associate input variables with a designated class label. Commonly employed classification loss functions include max likelihood and cross entropy. Conversely, in regression, the aim is to predict a real-valued quantity. Notable loss functions applied in regression for deep neural networks encompass Mean Square Error (MSE) and Mean Absolute Error (MAE).

2.1.5 Optimizers

In the realm of deep learning, optimizers refer to algorithms designed to adjust a model's parameters throughout training with the aim of minimizing a specified loss function. These algorithms facilitate the learning process of neural networks by iteratively updating weights and biases. Widely used optimizers include Stochastic Gradient Descent (SGD), Adam, and RMSprop, each characterized by distinct update rules, learning rates, and momentum, contributing to the quest for optimal model parameters to enhance performance.

Optimizers, fundamentally, are optimization methods crucial for enhancing the performance of a deep learning model. These algorithms significantly influence both the accuracy and speed of training in the deep learning paradigm. The primary function of an optimizer is to modify the weights after each epoch during the training of a deep learning model, ultimately minimizing the loss function. Acting as a function or algorithm, an optimizer adjusts attributes of the neural network, such as weights and learning rates, thereby aiding in the reduction of overall loss and the enhancement of accuracy.

Selecting the appropriate weights for a deep learning model, given the multitude of parameters involved, presents a formidable challenge. Consequently, the choice of a suitable optimization algorithm becomes imperative for optimal model performance in specific applications. Therefore, a foundational understanding of these machine learning algorithms is

essential for data scientists before delving deeply into the field.

While various optimizers can be employed in machine learning models to alter weights and learning rates, the optimal choice depends on the specific application. Although, as a beginner, the notion of trying multiple possibilities and selecting the one yielding the best results might seem plausible, it becomes evident that this approach is impractical, especially when dealing with substantial volumes of data. Randomly selecting an algorithm can be likened to gambling with valuable time, a realization that becomes apparent sooner or later in the learning journey.

2.2 Federated Learning

Federated learning, also recognized as collaborative learning, constitutes a machine learning strategy focused on training an algorithm across diverse local datasets situated in decentralized edge nodes or servers, all without the explicit exchange of parameters and other data information. This approach diverges from traditional centralized machine learning methodologies, where the transfer of local data is necessitated to a central device (server). Federated learning already offers substantial advantages compared to conventional machine learning techniques, particularly addressing privacy concerns.

In a typical federated learning setup, a central server is interconnected with numerous end devices. During the training process, each device refines its local model using its individual dataset and communicates solely the updated parameters to the server. Subsequently, for the commencement of the subsequent training cycle, the server disseminates the latest model parameters to the local devices after consolidating and computing a fresh global model. The topology of federated learning systems is used to classify them:

- **Centralized Federated Learning:** In centralized federated learning systems, a central server assumes the responsibility of orchestrating the training process and coordinating all participating nodes. However, the reliance on a single unit for updates from all local devices introduces a potential point of failure, as the server may become a bottleneck in the system.

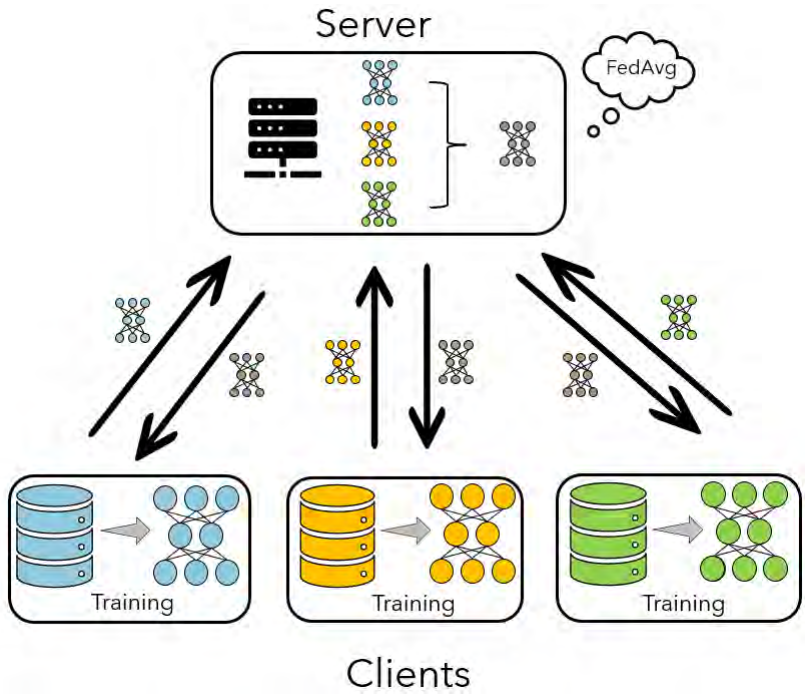


Figure 2.3: Centralized federated learning

- **Decentralized federated learning:** In decentralized federated learning [8], also referred to as peer-to-peer distributed learning, the absence of a central server is notable. Instead, the communication occurs exclusively in a peer-to-peer manner among the local nodes, facilitating the exchange of updates. In this configuration, the topology diverges from the star graph typical of server-client architecture and is instead depicted as a connected graph of nodes representing the clients. An edge in this graph signifies a communication channel between the clients, as illustrated in Figure 2.4.

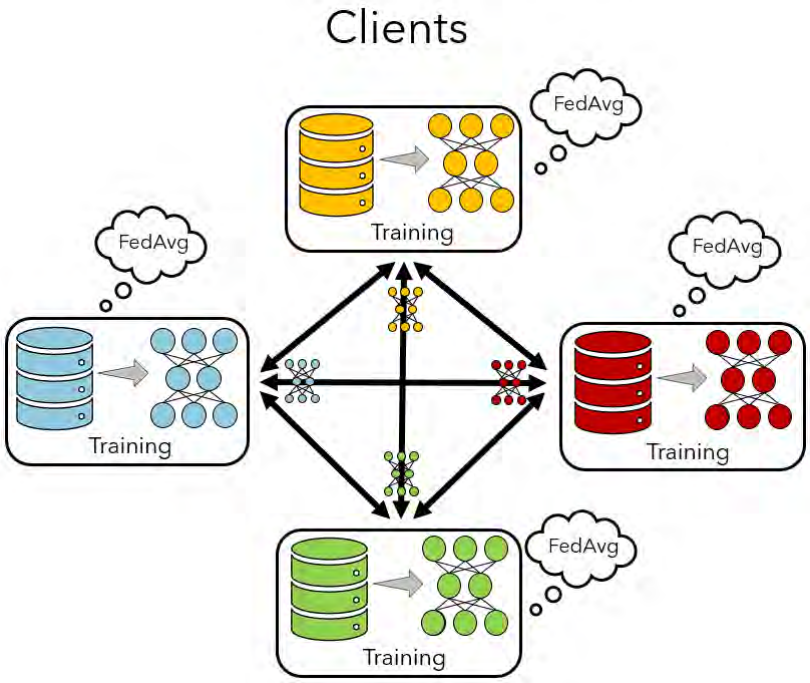


Figure 2.4: Decentralized federated learning

2.2.1 Benefits of Federated Learning

Federated learning offers significant advantages [9], particularly in preserving data privacy. Unlike conventional machine learning approaches that involve collecting and transmitting data to a central location, a process fraught with privacy and security concerns, federated learning operates by keeping the data on the device itself. This inherent approach ensures that data remains private and secure, thereby mitigating the risk of potential data breaches. Furthermore, federated learning achieves a reduction in the exchanged data between devices. This reduction stems from the fact that only the trained model is transmitted back to the server, avoiding the transfer of raw data. This not only diminishes bandwidth requirements but also minimizes the time needed for data transfer.

Beyond enhancing privacy and security, federated learning introduces greater flexibility into the model training process. In traditional machine learning, the model's adaptability to new data is constrained by using a fixed dataset. Federated learning, on the other hand, facilitates training a model on a dynamic dataset as soon as new data becomes available. This flexibility empowers the model to adapt and improve its accuracy over time. In summary, federated learning provides a trifecta of benefits: ensuring privacy and security by keeping data on devices, reducing data transfer needs, and enhancing model training by accommodating real-time updates on dynamic datasets.

2.2.2 Use Cases of Federated Learning

Federated learning finds diverse applications spanning various industries such as healthcare, finance, automotive, and personalized medicine. In the healthcare sector, the utilization of federated learning allows the training of models using patient data while upholding privacy standards. This approach enables healthcare professionals to enhance their models, identifying trends and patterns within patient data for improved diagnostic and treatment outcomes [10].

Within the financial industry, federated learning proves instrumental in training fraud detection models without exposing sensitive financial data to potential threats from malicious entities. In the automotive sector, the application of federated learning is valuable for training autonomous driving models directly on data from vehicles, eliminating the need to transfer substantial amounts of data to central servers.

Moreover, federated learning holds promise in the realm of personalized medicine. Unlike traditional machine learning, which relies on fixed datasets that may not fully encapsulate individual characteristics, federated learning enables the training of models using specific individual data. This approach contributes to heightened accuracy in diagnosis and treatment tailored to an individual's unique characteristics.

2.3 Clustering

Clustering [11] stands out as a crusial component in addressing the challenges of unsupervised learning, wherein the objective is to unveil patterns within unlabeled datasets. In essence, a cluster represents a set of elements that exhibit a degree of ”similarity” among themselves while also possessing distinctive characteristics from elements in other clusters. Clustering manifests in two primary forms: partitional and hierarchical. Hierarchical algorithms aim to identify clusters that evolve progressively from the initial state, with agglomerative algorithms building clusters from the bottom up, and divisive algorithms operating top-down. Conversely, partitional algorithms are employed to simultaneously discover all clusters within the dataset. Agglomerative algorithms initiate with individual elements as separate clusters, progressively amalgamating them into larger clusters. In contrast, partitional algorithms commence with the entire dataset, systematically partitioning it into successive components or smaller clusters.

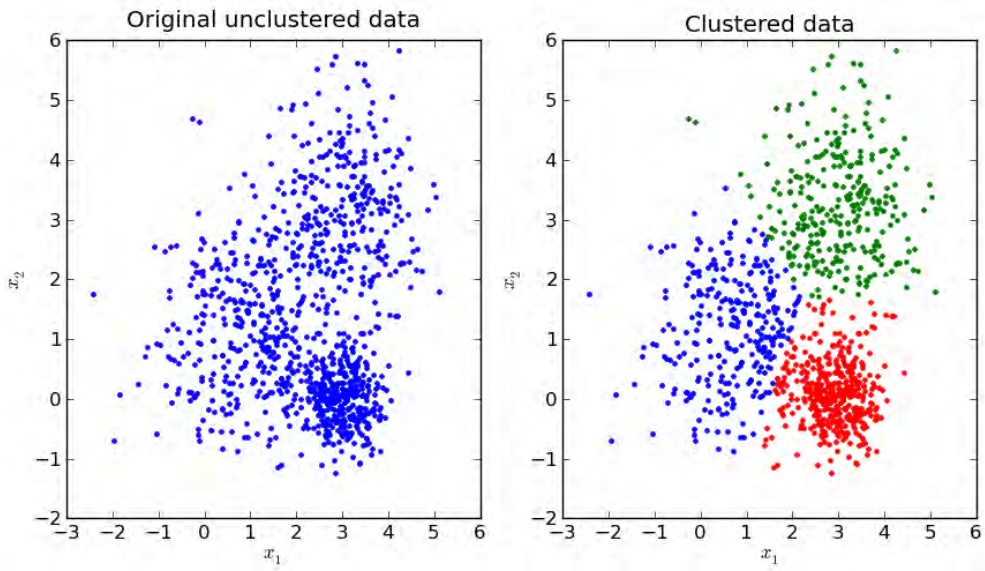


Figure 2.5: Clustering

2.4 Centrality

Centrality [12] is a fundamental concept in network analysis, offering a lens through which we can quantify the importance, influence, or prominence of individual nodes within a network. Networks, whether they represent social interactions, information flow, or infrastructural connections, are comprised of nodes and edges. Centrality measures help us identify key nodes that play crucial roles in shaping the overall structure and functioning of the network.

One of the simplest centrality measures is **Degree Centrality**, which focuses on the number of connections a node has. A high degree centrality suggests that a node is directly linked to many others, indicating potential significance in terms of information transfer, influence, or resource sharing.

Closeness Centrality emphasizes the proximity of a node to others in the network. Nodes with high closeness centrality can be seen as central in terms of efficient communication, as they are closer, on average, to all other nodes in the network.

Betweenness Centrality identifies nodes that act as crucial intermediaries in facilitating communication or interaction between other nodes. A high betweenness centrality suggests that a node lies on many of the shortest paths between other nodes, often acting as a bridge or bottleneck.

Eigenvector Centrality takes into account both the quantity and quality of connections. Nodes with high eigenvector centrality are not only connected to many others but are also connected to nodes that are themselves well-connected, conveying a sense of prestige or influence.

PageRank, originally developed by Google for ranking web pages, measures a node's importance based on both its direct connections and the importance of nodes that link to it. This algorithm considers the global structure of the network, rewarding nodes that are linked to by other important nodes.

Chapter 3

Proposed Methods

In the context of a fully distributed wireless network, relying on a single server introduces significant challenges, particularly in terms of server overhead. Additionally, implementing a device-to-device (D2D) protocol exacerbates issues related to the transmission of numerous messages.

To address these challenges, we propose a novel solution: the division of the entire network into subnets, each governed by a designated leader. This approach streamlines communication and mitigates the burden on servers, promoting more efficient network management.

Within each subnet, nodes exclusively communicate with their assigned leader to obtain updated weights for their neural networks. Simultaneously, leaders engage in inter-subnet communication to exchange weights and gather comprehensive information about the entire network. This decentralized structure significantly reduces the overall message volume.

Moreover, the deployment of numerous distributed servers (leaders) throughout the network ensures that the system remains agile and resilient. This approach eliminates the risk of overwhelming overhead on a central server, enhancing the scalability and robustness of the network.

The subsequent sections detail the algorithms employed to implement this innovative architecture, encompassing the clustering algorithm for subnet creation, the leader election algorithm to designate leaders within each subnet, and additional methods aimed at optimizing the network's efficiency by meticulously capturing the neural network intricacies of each node.

3.1 Wireless Network Architecture

3.1.1 Network

To enhance the clarity of algorithms explanations, we will refer to Figure 3.1, which illustrates an example network. This visual aid is intended to provide a clearer and more effective elucidation of the algorithms in question.

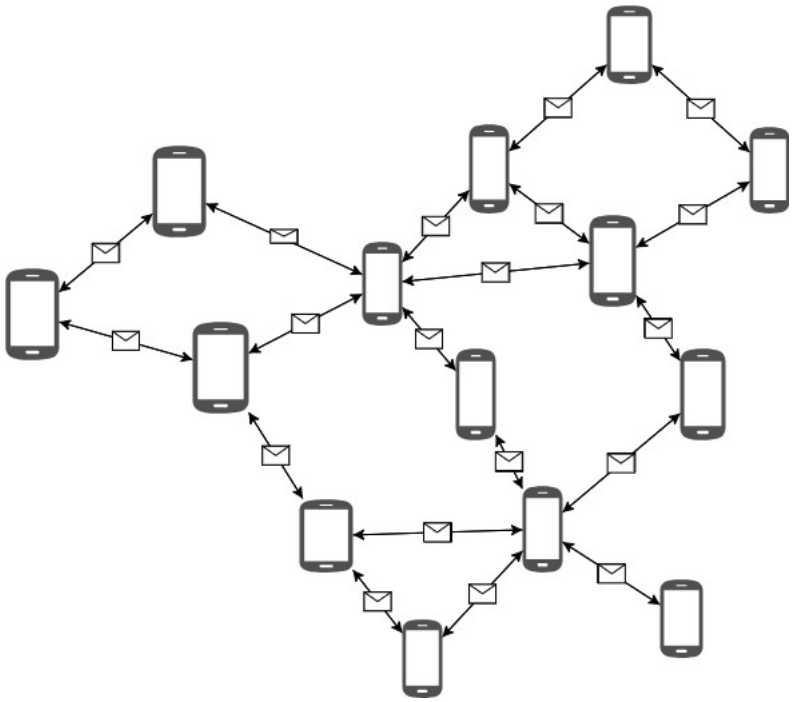


Figure 3.1: Example of a wireless network

3.1.2 Watts–Strogatz Model

To be able to generate these types of graphs, we used the Watts–Strogatz model [13]. This model introduced by Duncan J. Watts and Steven Strogatz in 1998, is a generative model for small-world networks. It constructs networks with local clustering and short average path lengths—features commonly observed in real-world networks.

The model begins with a regular lattice structure, where nodes connect to their k nearest neighbors in a ring-like fashion. Although this regular lattice exhibits a high local clustering coefficient, it also has long average path lengths. To introduce small-world properties, the model incorporates a rewiring parameter p ($0 \leq p \leq 1$). With a probability p, each edge in the network is rewired to connect to a random node instead of its regular lattice neighbor,

introducing shortcuts and reducing average path lengths.

The parameter k represents the number of nearest neighbors to which each node is initially connected in the regular lattice. Increasing k enhances the local clustering coefficient but also increases the average path length. As p increases from 0 to 1, the network transitions from a regular lattice to a small-world network. At $p = 0$, the network is a regular lattice, and at $p = 1$, it becomes a completely random graph.

The Watts–Strogatz model captures the small-world phenomenon, achieving a relatively short average path length between nodes in networks with high local clustering. This is accomplished by introducing a small number of random connections that act as shortcuts. The model has been applied to study various real-world networks, including social, biological, and technological networks, providing insights into the emergence of small-world properties in complex systems.

Overall, the Watts–Strogatz model is a valuable tool for understanding the interplay between local structure and global connectivity in networks. It highlights the balance between high local clustering and short average path lengths observed in many real-world systems. In our experiment, we created a graph with defined parameters ($k = 3$, $p = 0.5$) and a total of 14 nodes. The result of using the Watts–Strogatz model is illustrated in Fig. 3.3.

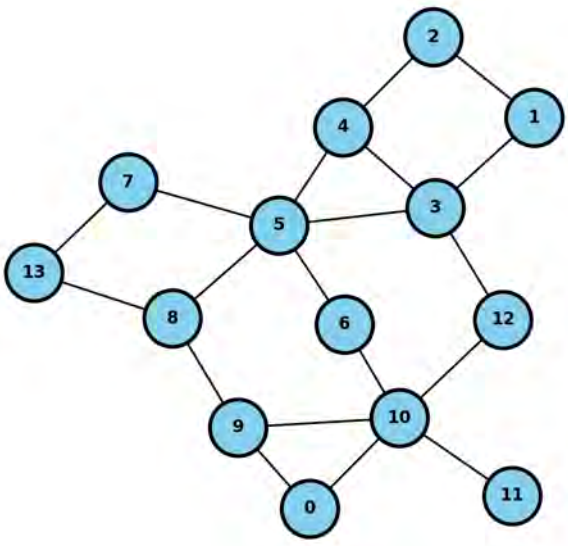


Figure 3.2: Result of Watts–Strogatz Model

3.2 Louvain Clustering Algorithm

The algorithm used to divide the network into subgroups was the Louvain clustering algorithm [14]. This algorithm introduced by Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre in 2008, is a popular and efficient method for detecting communities or clusters in complex networks. This algorithm is particularly well-suited for large-scale networks and has found applications in various fields, including social network analysis, biological network analysis, and information retrieval.

3.2.1 Overview

The algorithm used to divide the network into subgroups was the Louvain algorithm. This algorithm is a modularity-based approach, aiming to identify communities within a network by optimizing a quality function known as modularity. Modularity, denoted as Q, measures the strength of the division of a network into modules or communities based on the density of edges within communities compared to random expectations.

The modularity Q is defined as follows:

$$Q = \frac{1}{2m} \sum_{ij} \left(A_{ij} - \frac{k_i k_j}{2m} \right) \delta(c_i, c_j)$$

where:

- A_{ij} is the weight of the edge between nodes i and j,
- k_i and k_j are the sums of the weights of the edges attached to nodes i and j, respectively,
- The communities that nodes i and j belong to are c_i and c_j.
- m is the sum of all edge weights in the network, and
- $\delta(c_i, c_j)$ is the Kronecker delta function, which equals 1 if $c_i = c_j$ and 0 otherwise.

Phase One (Modularity Optimization at Node Level)

In Phase One, the algorithm iteratively moves nodes between communities to maximize the gain in modularity. This is achieved by optimizing the following expression for each node i:

$$\Delta Q = \sum_{in} \left(\frac{\Sigma_{in} + k_{i,in}}{2m} - \left(\frac{\Sigma_{tot} + k_i}{2m} \right)^2 \right)$$

where:

- Σ_{in} is the sum of the weights of the edges inside the community to which node i belongs,
- $k_{i,in}$ is the sum of the weights of the edges from node i to nodes in the same community,
- Σ_{tot} is the sum of the weights of the edges inside the entire network, and
- k_i is the sum of the weights of the edges attached to node i.

The process continues until no further improvement in modularity can be achieved at the individual node level.

Phase Two (Aggregate Communities)

In Phase Two, the network is represented at a higher level, and the modularity optimization process is repeated at this level, effectively identifying and refining the communities at a larger scale.

These mathematical expressions capture the essence of the Louvain clustering algorithm, providing a formal representation of the modularity-based optimization used for community detection.

This two-phase approach allows the Louvain algorithm to efficiently identify communities in large networks while maintaining a good balance between computation time and the quality of the identified communities.

3.2.2 Advantages

1. **Efficiency**: The Louvain algorithm is known for its efficiency, making it applicable to large-scale networks with millions of nodes and edges.
2. **Resolution Parameter**: The algorithm incorporates a resolution parameter that enables the user to control the size of the identified communities, providing flexibility in tuning the granularity of the clustering.

3. **Community Hierarchy**: The Louvain algorithm naturally produces a hierarchical structure of communities, revealing insights into the nested organization of nodes within the network.

4. **Widespread Adoption**: Due to its speed and effectiveness, the Louvain algorithm has been widely adopted in various domains for community detection tasks.

In conclusion, the Louvain clustering algorithm stands as a robust and efficient tool for uncovering the inherent community structure in complex networks, making it a valuable asset for researchers and practitioners seeking to gain insights into the organization of interconnected data. Using the the Louvain Algorithm in Fig 3.2 we receive that result:

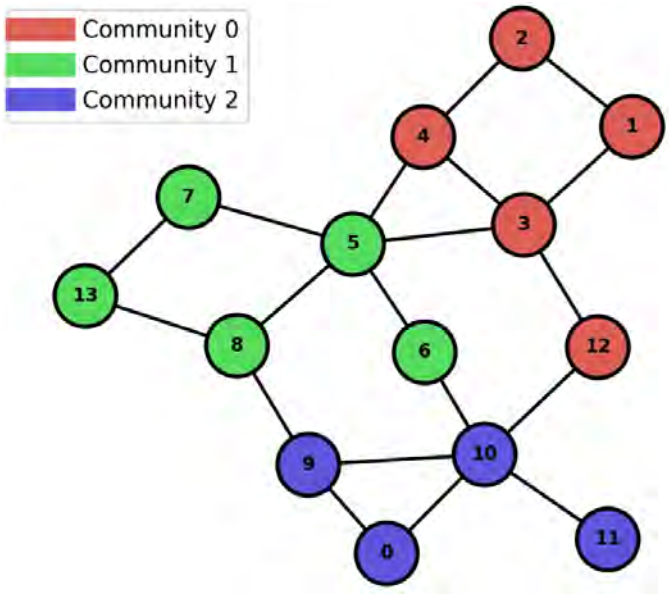


Figure 3.3: Result of Louvain Clusterin Algorithm

3.3 Closeness Centrality Algorithm

The algorithm used to elect leader of each group of the network was Closeness centrality [15]. This algorithm is a metric used to quantify the centrality of a node within a network based on the reciprocal of its average geodesic distance to all other nodes. A node with high closeness centrality is, on average, closer to all other nodes in the network.

The closeness centrality (C_i) of a node i in a connected graph is defined as:

$$C_i = \frac{1}{\sum_j d(i,j)}$$

where: - $d(i, j)$ is the geodesic distance (shortest path length) between nodes i and j.

The numerator of the expression represents the sum of the reciprocals of the gaps between node i and every other node in the network. The larger the closeness centrality, the closer the node is, on average, to all other nodes.

3.3.1 Normalized Closeness Centrality

In some cases, it is useful to normalize closeness centrality to account for differences in network size and connectedness. The normalized closeness centrality (C_{norm}) is given by:

$$C_{\text{norm},i} = \frac{N-1}{\sum_j d(i,j)}$$

where N is the network's total node count.

3.3.2 Applications of Closeness Centrality

Closeness centrality is employed in various fields, including social network analysis, transportation planning, and information flow analysis. In social networks, nodes with high closeness centrality are considered to have efficient access to information and are often crucial for the overall network connectivity. In transportation networks, closeness centrality helps identify locations that serve as efficient hubs for accessibility.

In conclusion, closeness centrality provides valuable insights into the proximity of nodes within a network, shedding light on their importance in facilitating information flow and connectivity. Finally, using the Closeness Centrality algorithm, we define the leaders of each group so that the experiment can be executed correctly, the result is shown in the Figure 3.5, where the leaders elected are shown in gold.

3.4 Federated Learning Process

The federated learning algorithm can be briefly presented by the following methodical steps:

1. During the first round of the procedure the entirety of the nodes of our network implements local training with a random chunk of data.

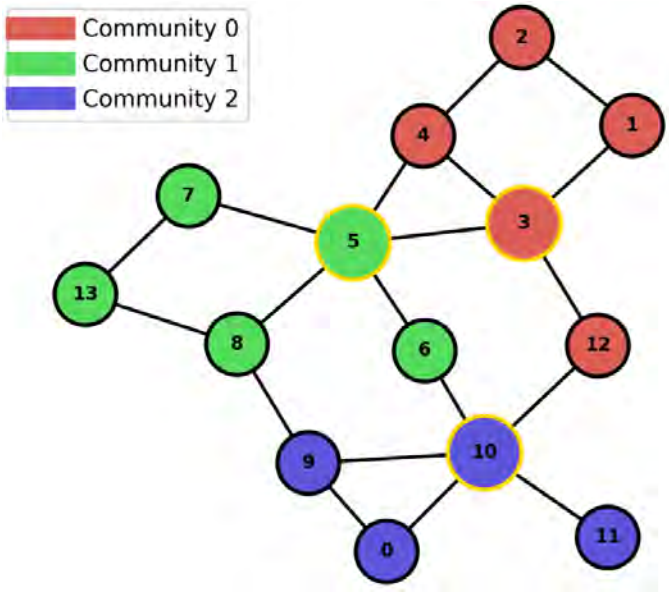


Figure 3.4: Result of Closeness Centrality

2. In the second and every other run, before the training begins each node send the trainable parameters of the previous round to their clusterhead, which will be operating as a local server.

3. Within each cluster, local servers gather trainable parameters from all participating clients during training, creating a package that encapsulates the midpoint of the weights within the group. Subsequently, an aggregate average is computed through communication among cluster heads. Employing the backbone method, servers exchange the previously assembled packets.

4. Upon recalculating the average from the newly received packets shared by other leaders, cluster heads disseminate it to their respective cluster members, only involving the nodes that actively participated. Consequently, in the subsequent learning round, each node incorporates precisely the same weights into its machine learning model.

5. The decision to conclude the algorithm occurs once the accuracy of the majority of models has reached a satisfactory level, signifying the attainment of convergence.

3.4.1 Node Participation Protocols

Following the FL heuristic, the designated nodes initiate the training process of the machine learning model in collaboration with clusterheads, acting as local servers. These clus-

terheads exchange training variables not only among themselves but also with their respective cluster members. Given the challenges posed by extensive networks, large datasets, and intricate ML models, any means to minimize unnecessary and incongruent communication between nodes holds significant value. Consequently, we have identified two potential solutions within the framework of Node Participation Protocols to both mitigate costs and enhance the precision of the ultimate results.

- **Random participation of nodes**

  The initial and relatively straightforward approach to achieve this is by assigning random nodes to participate. Subsequently, a percentage of nodes within each cluster is chosen based on their weights to undergo a training round. This method expedites the learning process and reduces the expense associated with message exchange. However, it is evident that over time, this strategy may lead to quality issues. The challenge lies in the uncertainty of identifying which node, in which round, can provide the most significant assistance and contribute effectively to the training process.

- **Best Accurancy participation of nodes**

  An alternative approach to mitigate memory reduction involves employing the best accuracy algorithm, which follows a straightforward logic. In this method, when the leader of each group receives the weights of other nodes within the group, they also receive the accuracy metrics captured by the neural network of each node. The leader then makes decisions based on selecting nodes with the highest accuracy to participate in the training process.

3.4.2 Federated Averaging Algorithm

Each node, denoted as i, exclusively possesses access to its local dataset D_i. All nodes share a common machine learning model class, aiming to minimize the objective function $F(x)$ defined as:

$$F(x) = \frac{1}{n} \sum_{i=1}^{n} F_i(x)$$

Here, x represents the model parameters (weights), and $F_i(x)$ is the local objective function. The Federated Averaging Algorithm [16] involves two key steps:

- **Federation and Averaging within Each Cluster:** At each round and within each cluster, nodes that adhere to the node participation protocols outlined earlier engage in federation. They individually train their local models, sharing their parameters (weights) with the selected clusterhead. The clusterhead accumulates all local updates, including its own, and performs the averaging process. Subsequently, the cluster nodes synchronize with the new parameters and continue training with the updated model. In this context, the clusterhead essentially functions as a local server within the cluster.

- **Global Federation and Averaging:** Global averaging occurs between clusters. Specifically, every time that the clusterhead has all the parameters of the others nodes, the clusterheads from each cluster communicate with one another, exchanging their parameters. Global aggregation takes place, computing new global parameters, which are then broadcasted back to every clusterhead to initiate the next round of training.

Chapter 4

Purely Distributed FL Simulator

4.1 Introduction

In order to be able to test the new techniques to reduce the communication of the nodes, a simulator was created. Below we will analyse what was used to implement it, how it works and what new algorithms it uses to reduce the messages.

4.2 Tools and Platforms for the implementation

4.2.1 Docker

Docker [17] is a platform that streamlines the development, deployment, and management of software applications. It employs containerization technology to encapsulate applications and their dependencies within isolated and lightweight environments known as containers. Docker containers are derived from docker images, which serve as the architectural plans for an application's deployment environment. Developers utilize docker images to bundle their applications along with essential libraries, configurations, and runtime environments, ensuring consistent behavior across various host machines. The creation process of a docker image is described by Dockerfiles, which are text-based configuration files providing a set of instructions to construct an image layer by layer. Consequently, as indicated in Figure 4.1, a Dockerfile constructs a docker image that, upon deployment, functions as a docker container.

Docker provides a level of abstraction that separates the operating system and underlying hardware, enhancing the flexibility of software development. This abstraction allows

Figure 4.1: Dockerfile to Image to Container

developers to create software that can run consistently across various environments. Notably, Docker is lightweight, optimizing operational efficiency. Containers operate as distinct processes in user space, sharing the kernel of the host's operating system, as illustrated in Figure 4.2. This design surpasses the performance of virtual machines.

Docker proves particularly advantageous for microservices architectures and continuous integration/continuous deployment (CI/CD) pipelines due to its superior container startup times and resource utilization. Moreover, Docker supports layering, enabling the reuse of a base image in different containers. This not only reduces storage requirements but also minimizes network bandwidth usage.

In addition to its well-known features, Docker supports orchestration, a less-publicized capability. Similar to Kubernetes, Docker facilitates deployment on clusters, where software is effectively managed and scaled across container clusters.

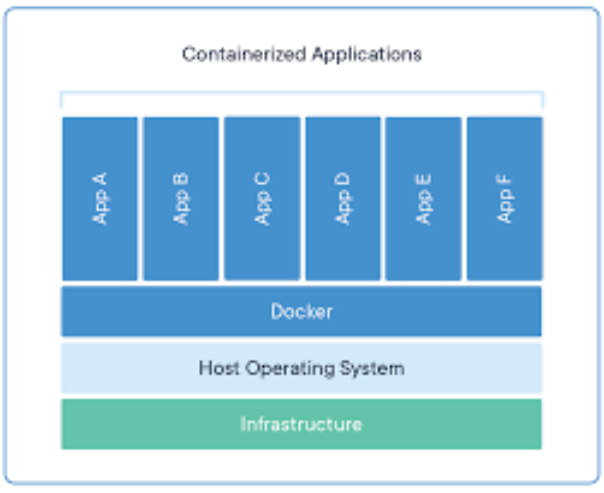


Figure 4.2: Single Machine Hosting Multiple Docker Containers

Docker virtual network

Docker virtual networks [18] are a way to isolate and manage network communication between containers. They provide containers with their own network environment, enabling secure and controlled communication. You can create custom networks, connect containers to them, and even control external connectivity. Docker virtual networks are crucial for building and orchestrating containerized applications. The figure below depicts the functionality of a virtual network connecting two containers.

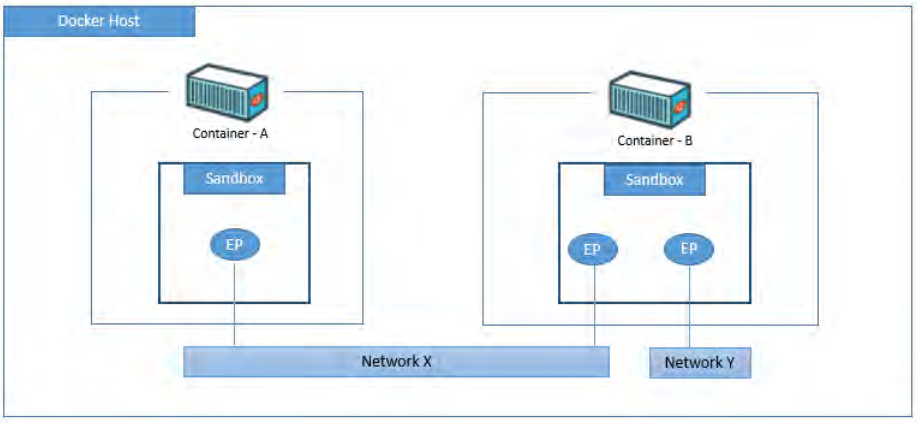


Figure 4.3: Example of a Docker virtual network

4.2.2 MySQL Database

Databases serve as the fundamental storage repository for all software applications. For instance, whenever an individual performs a web search, logs into an account, or completes a transaction, a database system captures and stores the information for future retrieval.

In the case of a relational database [19], data is organized into distinct tables instead of being stored in a single expansive repository. The physical file structure is optimized for speed, while the logical data model incorporates objects such as data tables, views, rows, and columns, creating a flexible programming environment. Establishing rules governing relationships between various data fields such as one to one, one to many, unique, required, or optional and establishing ”pointers” between different tables ensures data consistency. A well-designed database enforces these rules, preventing the occurrence of inconsistent, duplicated, orphaned, outdated, or missing data in your application.

The acronym ”SQL” in ”MySQL” stands for ”Structured Query Language,” which is the most widely used standardized language for accessing databases. Depending on the program-

ming environment, one may interact with SQL directly (e.g., for report generation), embed SQL statements into code written in another language, or utilize a language-specific API that conceals the SQL syntax.

4.2.3 MongoDB Database

MongoDB [20] stands out as an open-source NoSQL database management program, offering a compelling alternative to traditional relational databases. The term ”NoSQL,” signifying ”Not only SQL,” reflects its versatility in handling large sets of distributed data. MongoDB excels as a tool for managing document-oriented information, facilitating the storage and retrieval of data.

This database is particularly valuable for organizations dealing with high-volume data storage requirements, enabling them to efficiently manage extensive datasets without compromising performance. MongoDB finds widespread use due to its capabilities in ad-hoc queries, indexing, load balancing, aggregation and various other features.

In contrast to the structured approach of Structured Query Language (SQL) used in relational databases, MongoDB employs a different architecture. It comprises collections and documents instead of tables and rows. Documents, composed of key-value pairs, serve as MongoDB’s fundamental unit of data. Collections, akin to SQL tables, house sets of documents. MongoDB boasts compatibility with numerous programming languages.

4.2.4 Docker SDK

The Docker SDK [21] provides a comprehensive set of tools and APIs designed for the creation and management of Docker containers. This toolkit empowers users to interact with Docker containers programmatically, enabling seamless integration and manipulation of containerized environments.

With the Docker SDK, users can effortlessly create, customize, and control containers, offering a versatile solution for container orchestration. This toolkit supports various programming languages, catering to diverse development preferences.

In essence, the Docker SDK is a robust resource for anyone seeking to efficiently create and manage Docker containers, facilitating a streamlined and adaptable approach to containerization.

4.2.5 Multiproccessing

The multiprocessing library in Python serves as a powerful tool for parallel computing, allowing developers to design applications that run concurrently, leveraging the capabilities of multi-core processors. By providing a user-friendly and intuitive interface, this library simplifies the creation and coordination of independent processes, enabling efficient parallel execution.

Whether handling computationally intensive tasks, optimizing performance, or enhancing responsiveness, multiprocessing proves invaluable. Its features, including process pools and communication mechanisms, empower developers to create scalable and responsive applications that take full advantage of modern computing architectures. With multiprocessing, the complexity of parallel programming is mitigated, making it an essential resource for optimizing computational workflows.

4.2.6 ZeroMQ

ZMQ, short for ZeroMQ, [22] is a high-performance asynchronous messaging library and framework that provides sockets for building distributed and networked applications. It's designed to make building scalable, distributed, and concurrent applications easier by providing abstractions for various communication patterns. ZeroMQ is often referred to as a "messaging kernel" because it enables communication between different parts of an application, even across different programming languages and platforms. Key features and concepts of ZeroMQ include:

Asynchronous Messaging: ZeroMQ provides a message-passing mechanism that allows components of a distributed application to communicate asynchronously. This means that sender and receiver do not need to be synchronized, and messages can be sent and received without blocking.

Socket Types: ZeroMQ offers different socket types, each tailored to a specific communication pattern. Some common socket types include:

- REQ-REP: Request-Reply pattern.
- PUB-SUB: Publish-Subscribe pattern.
- PUSH-PULL: Load balancing and fan-out.

- PAIR: Peer-to-peer communication.

Cross-Language Support: ZeroMQ supports multiple programming languages, including C, C++, Python, Java, and more. This allows you to build distributed systems with components written in different languages.

Scalability: ZeroMQ is designed to handle high loads and can be used for building scalable and high-performance applications. It's suitable for use in both small-scale and large-scale systems.

4.2.7 MessagePack

In Python, the MessagePack [23] module serves as a speedy and efficient binary serialization format. Its purpose is to offer a more space-efficient alternative to JSON, making it highly valuable in situations where data size and the speed of serialization and deserialization are critical factors.

Once you've incorporated the module, you gain access to functions that facilitate encoding and decoding of data. For instance, you can utilize msgpack.packb() to convert a Python object into a bytes object through serialization, and msgpack.unpackb() to reverse this process by deserializing a bytes object back into a Python object.

4.2.8 NumPy

NumPy [24], a Python library dedicated to numerical and scientific computing, stands out for its open-source nature. Specifically designed for the efficient handling of multi-dimensional arrays, NumPy optimizes operations on these arrays, providing a powerful data structure for storing numerical data, which proves instrumental for mathematical and statistical manipulations. Beyond array operations, NumPy incorporates optimized linear algebra functionalities, contributing to enhanced performance and memory efficiency. This library's versatility and robust performance have solidified its widespread adoption in both data science and machine learning.

A notable attribute of NumPy is its seamless integration with other essential libraries, including SciPy, Matplotlib, Pandas, PyTorch, and TensorFlow. This integrative capacity strengthens NumPy's role in a broader ecosystem for data analysis, visualization, and computing. Researchers and developers benefit from NumPy's capabilities, enabling them to

seamlessly work with numerical data and conduct complex computations within the Python programming language.

4.2.9 Pandas

Pandas [25], an open-source Python library extensively utilized by data scientists and developers, stands out for its robust capabilities in data processing and analysis [21]. Built upon NumPy, Pandas introduces two primary data structures: the Series and the DataFrame. The Series represents a single-dimensional labeled array accommodating diverse data types, while the DataFrame is a two-dimensional table-like structure, akin to a spreadsheet or an SQL table, organizing data into columns and rows. Its significance in machine learning lies in its facilitation of efficient data cleaning, transformation, and exploration. Pandas streamlines tasks such as data preprocessing, handling missing data, dataset aggregation, filtering, grouping, and data visualization with tools like Matplotlib.

An outstanding attribute of Pandas is its adaptability in loading data from various file formats, coupled with its capability to execute intricate mathematical and statistical operations on substantial datasets. Notably, Pandas simplifies the data analysis workflow, creating optimal conditions for machine learning. It adeptly manages large datasets through lazy evaluation and memory optimization techniques. Additionally, Pandas offers versatile indexing and selection methods, enabling users to access, manipulate, and analyze data effectively. Whether engaging in tasks such as loading data from diverse file formats, executing complex data operations, or generating summary statistics, Pandas streamlines the entire data analysis process, empowering users to extract valuable insights from their datasets.

4.2.10 TensorFlow and Keras

TensorFlow [26], an open-source end-to-end machine learning (ML) platform, empowers users to develop ML models and deploy them across diverse devices, spanning from servers and edge devices to the cloud. It facilitates data automation, model tracking, performance monitoring, and the processes of model retraining and inference. TensorFlow is widely employed for both the training and inference of deep neural networks, supporting implementation in Python, JavaScript, C++, and Java.

Keras, integrated into TensorFlow as a high-level API, serves as an interface for approaching and resolving ML problems with a primary focus on deep learning. It presents

a user-friendly and highly productive interface, covering every facet of the ML workflow. Keras encompasses a broad spectrum of features, spanning from data processing and hyperparameter tuning to deployment, and is designed to enable swift experimentation [27].

KerasTuner [28] emerges as an accessible and scalable hyperparameter optimization framework, adeptly exploring tuning hyperparameters within a specified search space. KerasTuner can employ pre-implemented search algorithms or accommodate the integration of experimental algorithms as extensions, unveiling optimal hyperparameters tailored to a specific model and dataset. Notably, KerasTuner seamlessly interfaces with Python and TensorFlow.

4.2.11 Threading

The threading module in Python [29] offers a versatile framework for managing threads, enabling developers to execute multiple threads concurrently—each serving as a smaller unit of a process. Threading proves invaluable for parallelizing tasks and elevating the responsiveness of applications. This becomes especially advantageous in situations where concurrent execution of specific operations, such as I/O-bound tasks, is beneficial.

ThreadPoolExecutor

The ThreadPoolExecutor [30] is a class in the concurrent.futures module of Python that provides a high-level interface for concurrently executing callables using a pool of threads. It is particularly useful for managing and parallelizing asynchronous tasks in a multithreaded environment.

Key Features Thread Pool Management: ThreadPoolExecutor efficiently manages a pool of worker threads, allowing for the concurrent execution of tasks. The pool is created upon instantiation, and tasks can be submitted to the pool for execution.

Asynchronous Execution: Tasks submitted to the thread pool are executed asynchronously, meaning they can run concurrently without blocking the main program's execution.

Callable Execution: Tasks can be any callable object, such as functions or methods, and they are scheduled for execution in the background.

Concurrency Control: The number of worker threads in the pool can be controlled, allowing for fine-tuning of concurrency based on system resources and application requirements.

4.2.12 Networkx

NetworkX [31] stands as a robust Python library tailored for the creation, analysis, and visualization of complex networks or graphs. This library empowers developers with a comprehensive suite of tools and functions, facilitating the manipulation of graph data structures. Supporting both directed and undirected graphs, including multigraphs and bipartite graphs, NetworkX becomes an invaluable asset in domains like social network analysis, biology, and infrastructure modeling.

Key features of NetworkX encompass graph creation, manipulation, and extensive algorithms for tasks such as shortest path calculation, centrality measures, clustering, and community detection. Visualization tools allow for the creation of clear and informative graphical representations, often integrated with popular plotting libraries such as Matplotlib [32].

Moreover, NetworkX provides graph generators for creating synthetic graphs with specific characteristics, enhancing its utility in the study of various network models. Its seamless integration with other Python libraries, such as NumPy and SciPy, further boosts its computational capabilities.

4.3 Implementation

In the following section, we will look in depth at the methodology of the simulator's construction. Specifically, there are two implementations: the full Docker build and the hybrid approach. In the full Docker build, the entire program is executed inside Docker containers, while in the hybrid model, half of the program is executed locally on the computer and the databases are housed inside Docker containers. The hybrid approach was introduced to streamline the development process by eliminating the need for frequent creation and termination of Docker containers. In the coming subsections, we'll start by taking a closer look full Docker build. We will then go into an analysis of these two approaches, describing their subtle differences.

4.3.1 Docker Containers

Our implementation required four types of Docker containers. Two of them are the databases and the other two are scripts, written in python. Below we'll explain how each one works and what tools were needed to make them work properly.

1. MongoDB container
2. MySQL container
3. Controller container
4. Node container

4.3.2 MongoDB container

Structure

The primary purpose of the database is to store datasets, which are important for our neural networks. The structural design we implemented is elegantly simple. We started by creating a dedicated database inside the container, called ”Datasets”.

While the experiments were done with a dataset, we introduced a collection called cifar-10 , which hosts the dataset used by our neural network. This dataset is meticulously organized into twenty-four different segments, with twenty being allocated for neural network training and the remaining four being reserved for evaluation purposes. Each of these segments

contains a total of 2,500 images. This meticulous separation of the data serves the purpose of allowing each computational node to access and select random images during the training process, thus enhancing the robustness and adaptability of the network. If figure below we can see the structure of this container.

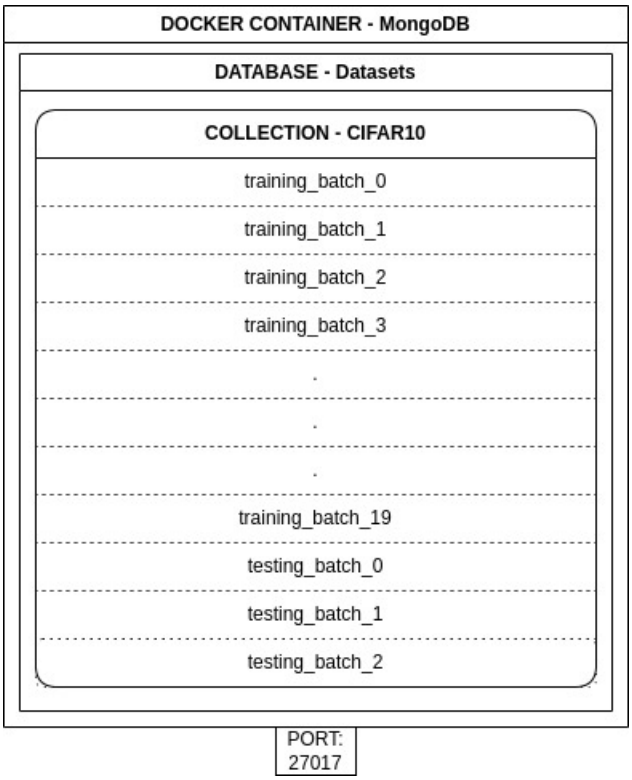


Figure 4.4: MongoDB Structure

docker-compose.yml file

The docker-compose.yml file specifies the use of the official MongoDB image from Docker Hub (image: mongo). It establishes a custom container name, 'mongodb_datasets,' ensuring clarity in identification. The container is configured to restart automatically (restart: always), ensuring persistent availability in case of unexpected disruptions.

To facilitate communication and networking, the container is connected to a dedicated virtual network named 'dataset_nodes' (networks: dataset_nodes). This network is specifically designed to enhance interaction between containers within the Docker environment.

The configuration also includes the definition of a volume named 'mongo-data' (volumes: mongo-data), utilizing the local driver to persistently store MongoDB data. The volume is mounted to the '/data/db' directory within the container, ensuring data persistence across

container restarts.

Port mapping (ports: - 27017:27017) allows external access to MongoDB on the host machine through port 27017, a commonly used port for MongoDB.

This detailed docker-compose.yml file not only encapsulates the necessary parameters for MongoDB but also streamlines the deployment and management of the container within a Dockerized environment.

```yaml
version: '3.8'

services:
  mongodb:
    image: mongo
    container_name: mongodb_datasets
    restart: always
    networks:
      - dataset_nodes
    volumes:
      - mongo-data:/data/db
    ports:
      - 27017:27017

volumes:
  mongo-data:
    name: mongo-data
    driver: local

networks:
  dataset_nodes:
    name: dataset_nodes
    driver: bridge
```

Figure 4.5: MongoDB compose file

4.3.3 MySQL container

Structure

The SQL database serves as a crucial repository for experiment information, primarily focusing on the structural aspects stored within the 'networks' and 'nodes' tables. The 'networks' table captures the essence of the experiment, holding details such as the total number of nodes, groups, repetitions, epochs per round, participation requirements, and participation rates.

Meanwhile, the 'nodes' table complements the network structure by featuring individual node information. This includes a node's unique identifier, its name, associated group and network IDs, and a designation of whether it functions as a leader within the network.

The experiment's outcomes find their home in the 'measurements' table. This table meticulously records data related to each node's performance, including iteration indices, message indices for sending and receiving, training and testing accurancies, losses, training time, and

the total length of messages.

To enhance user visibility into the experiment's progress, the 'logs' table captures and organizes all messages exchanged between nodes. Each log entry includes information such as the node involved, sender and receiver IDs, request type, message index, and message length.

By structuring the database in this manner, users gain comprehensive insights into the experiment's architecture, individual node performance, and the dynamic communication between nodes, facilitating a thorough examination of the experiment's progress. In figure below, we can see the structure of our MySQL database.

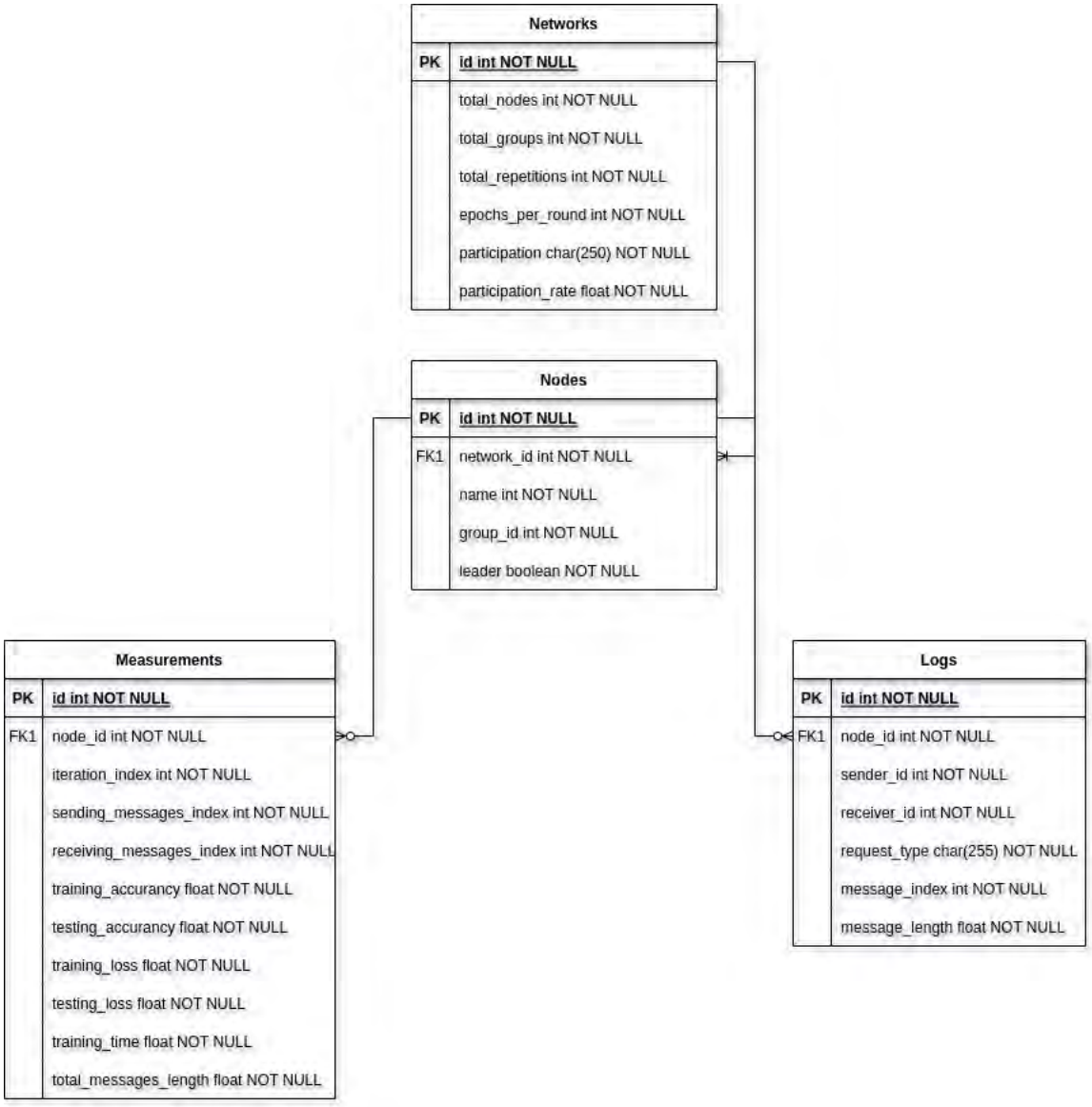


Figure 4.6: MySQL relational schema

docker-compose.yml file

The docker-compose.yml file outlines the deployment of our MySQL service using the official MySQL image from Docker Hub (image: 'mysql'). The service is uniquely identified by the container name 'mysqldb_statistics,' promoting easy recognition and management.

To enhance robustness, the container is configured to restart automatically with the "restart: always" directive, ensuring continuous availability in the face of unexpected disruptions or failures.

Communication between containers is facilitated through the establishment of a dedicated virtual network named "statistics_nodes". This network is designed to optimize interaction among containers within the Docker environment, fostering seamless data exchange.

The configuration further encompasses the definition of a volume named "init.sql". This volume, allows the initialization of the structure that we saw previously, using a custom SQL script. This approach ensures that the database is pre-configured with specified parameters upon container creation.

External access to the MySQL service is facilitated through port mapping ports: - 3306:3306. This configuration allows connections to the MySQL service on the host machine via port 3306, providing a standardized access point.

In summary, this detailed docker-compose.yml file encapsulates the essential parameters for the MySQL service, streamlining both deployment and ongoing container management within a Dockerized environment.

```yaml
version: '3.3'
services:
  mysql:
    image: 'mysql:latest'
    container_name: mysqldb_statistics
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
      MYSQL_USER: ${MYSQL_USER}
      MYSQL_PASSWORD: ${MYSQL_PASSWORD}
    ports:
      - '3306:3306'
    volumes:
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
    networks:
      - statistics_nodes

networks:
  statistics_nodes:
    name: statistics_nodes
    driver: bridge
```

Figure 4.7: MySQL compose file

4.3.4 Controller's Container

Inputs

The system is configured through ten input parameters, which influence the behavior of two classes: the controller and the graphGenerator. Upon starting the controller container, its class initiates and executes based on these input parameters, governing the evolution of the experiment.

The ten input parameters are as follows:

- **network_id:** A unique identifier enabling users to store and differentiate their experiments.
- **total_nodes:** The overall number of nodes present in the graph.
- **lower_limit:** The minimum number of nodes allowed in each group.
- **upper_limit:** The maximum number of nodes permitted in each group.
- **total_repetitions:** The total number of repetitions for message exchange.
- **epochs_per_round:** The number of training repetitions for each node before message exchange occurs.
- **participation:** The type of message exchange participation chosen by the leader of each group. Options include:
  - *Total Participation:* All nodes participate in federated learning.
  - *Random Participation:* A random percentage of nodes participate, based on the participation rate.
  - *Highest Accuracy Participation:* A percentage of nodes participate, determined by both the participation rate and the accuracy of their neural network.
- **participation_rate:** The percentage of nodes selected by the algorithm for message exchange participation.
- **one_hop_leader:** Specifies whether leaders of each group are separated by one or more than one hop in the graph.
- **is_docker:** Indicates whether the program runs locally or within a Docker container.

Controller Class

This class serves as the container's central hub, acting as the program's core orchestrator. It not only oversees user inputs but also dynamically generates the necessary containers and monitors the experiment's progress.

GraphGenerator Class

graphGenerator is the primary class responsible for creating and modifying graphs using the `networkx` library and other auxiliary tools. This class can generate two types of graphs:

1. **Random Graph:**

    - Takes three parameters: the total number of nodes in the graph and the lower and upper limits of nodes per group.
    - Applies the Louvain clustering algorithm to the generated graph, as mentioned earlier.
    - After creating the groups, it applies the closeness centrality algorithm mentioned previously.
    - Returns the graph to the controller with all the necessary information.

2. **One-Hop Leaders Graph:**

    - Follows the same logic as the random graph but creates smaller groups based on the provided lower and upper limits inputs.
    - Applies the closeness centrality algorithm to the generated graph.
    - Connects all leaders to each other to ensure they are one hop away.
    - Returns the graph to the controller with all the required information.

In both cases, the `graphGenerator` class employs algorithms to enhance the graph's structure and connectivity, providing the controller with a comprehensive graph containing all the relevant information.

Dockerlife

The controller container is central to our system, housing the essential 'controller.py' Python script along with 'graphGenerator.py.' To streamline deployment, we use a Dockerfile (Figure 4.9) with the following structure:

```
FROM python:3.9-slim-buster

# Set the working directory in the container
WORKDIR /app

# Update system packages
RUN apt-get update && \
    apt-get install -y python3 python3-pip && \
    rm -rf /var/lib/apt/lists/*

# Copy the requirements file
COPY requirements.txt .

# Install the Python dependencies
RUN pip3 install --no-cache-dir -r requirements.txt

# Copy the entire project directory into the container
COPY scripts/controller.py /app/
COPY scripts/graphGenerator.py /app/

ENTRYPOINT ["python3", "-u", "controller.py"]
```

Figure 4.8: Controller's Dockerfile

This Dockerfile encapsulates our Python-based application, installing dependencies and setting up the working directory. The 'controller.py' script serves as the entry point, ensuring seamless execution within the containerized environment.

4.3.5 Node's Container

These containers are generated in bulk by the controller at the start of the experiment and are terminated upon experiment completion. The controller generates as many of these containers as there are nodes in the user's input graph. These containers have three python files inside:

NodeSelector File

This script acts as the initialization point during container creation. It discerns whether the node functions as a leader or a regular node, relying on a predefined definition sourced from the container controller, as it oversees their creation. Following this determination, it identifies the fundamental class for execution.

leaderNode Class

When the node assumes the role of a group leader, a multithreaded script is generated, comprising four essential threads:

tcp_listener_thread: Responsible for receiving, forwarding, and storing messages specific to this node. To facilitate this communication, two zmq sockets are employed. The utilization of polling mechanisms with these sockets helps mitigate potential errors and ensures a smoother data exchange process.

To enhance the efficiency of the communication thread and prevent overloading the nodes, the system leverages the ThreadPoolExecutor library. This library is instrumental in managing concurrent tasks efficiently, allowing the node to handle multiple communication tasks concurrently without compromising performance. The ThreadPoolExecutor optimizes resource utilization and streamlines the handling of incoming messages, contributing to the overall responsiveness and reliability of the node's communication system.

training_thread: Focusing in neural network training using the TensorFlow library discussed earlier, this component not only updates the weights of the neural network when it receives data from the team leader, but also records the data in the MySQL database. Specifically, it writes to the "measurements" table at the end of each training cycle, capturing details about the performance of the neural network and additional information about message exchanges with other nodes. As a final step, it transmits its neural network weights to the leader, facilitating the ongoing iteration of federated learning.

group_info_thread: Tasked with maintaining information about nodes within the same group, its responsibilities extend to storing the weights transmitted by nodes. To compute the average, it uses the Federated Averaging Algorithm mentioned earlier, leveraging the functionality of the NumPy library. It then propagates the computed average of the group weights to other leaders.

The structural arrangement of the storage table for nodes within each group is visually depicted in the figure below. The stored values encompass the node's identifier (ID), a boolean indicating whether the node actively participates in federated learning. Additionally, there is an indicator specifying whether the node has transmitted its neural weights at a given time, enabling the application of the averaging algorithm once all nodes have contributed. The index denotes the number of messages the node has dispatched, accuracy measures the precision of its neural network, and finally, the neural weights are preserved for comprehensive

record-keeping.

leader_info_thread: It operates similarly to the group_info_thread, but it processes the messages it receives from the other leaders. This thread is responsible for sending the final average of the neural network weights from the entire network to the nodes of the group.

The structural arrangement of the storage table for leader nodes is visually depicted in the figure below. The stored values encompass the node's identifier (ID). Additionally, there is an indicator specifying whether the node has transmitted its neural weights at a given time, enabling the application of the averaging algorithm once all nodes have contributed. The index denotes the number of messages the node has dispatched and finally, the neural weights are preserved for comprehensive record-keeping.

Group's nodes information	
Value	Type
node_id	int
participate	boolean
has_send	boolean
index	int
accurancy	float
weights	list

Figure 4.9: storage table for each group's nodes

leader's nodes information	
Value	Type
node_id	int
has_send	boolean
index	int
weights	list

Figure 4.10: storage table for each leader's nodes

groupNode Class

Similarly, the groupNode class follows a parallel logic to the previous class, although it is differentiated by the including of two threads. The first thread, similar to that of the leader, is dedicated to the training model. In contrast, the second thread, responsible for the TCP listener, diverges in its approach to message processing, using a separate evaluation of received messages. These dual threads collectively constitute the necessary elements that are critical to the complete operation of the node.

Dockerfile

In conclusion, for the creation of the Docker image, presented below is the structured Dockerfile. This serves as a comprehensive blueprint for constructing the image efficiently

and ensuring the seamless deployment of our Python-based application.

```
FROM python:3.9-slim-buster

WORKDIR /app

RUN apt-get update && \
    apt-get install -y python3 python3-pip && \
    rm -rf /var/lib/apt/lists/*

COPY requirements.txt .

RUN pip3 install --no-cache-dir -r requirements.txt

# Copy the main.py file
COPY scripts/nodeSelectorDocker.py /app/

# Copy the entire classes directory
COPY scripts/classes/ /app/classes/

ENTRYPOINT ["python3", "-u", "nodeSelector.py"]
```

Figure 4.11: Node's Dockerfile

4.3.6 Communication between containers

Structure of messages

The structure of the messages that the containers communicate is a dictionary of a python and it has five values:

- **Sender_ID**: is the id of the node that sends the message.
- **Receiver_ID**: is the id of the node that receives the message.
- **Request_Type**: is the type of the message so the receiver understand the purpose of the message and decoded properly. There are seven types of messages:
  - NETWORK_TOPOLOGY: It's sent from the controller to the nodes, providing information on the topology of the graph.
  - START_TRAINING: It's sent from the controller to the nodes, to start the experiment from the nodes.
  - NETWORK_TERMINATION: It's sent from the controller to the nodes, for the termination of nodes.
  - GROUP_LEADER: It's sent from node to node, for sending the information of the neural network weights to the group leader.
  - LEADER_RESPONSE: It's sent from node to node, for the taking of the final results created by each leader.
  - LEADERS_COMMUNICATION: It's sent from node to node, for the exchange of messages between the leaders.
  - SIMULATION_TERMINATION: It's sent from the node to the controller for the complete its task.
  - NODE_ERROR: It's sent from node to controller, on the inefficiency of a node response.
- **Index**: ιs to keep the messages numbered, for better error prevention.
- **Data**: according to request type the data is different.

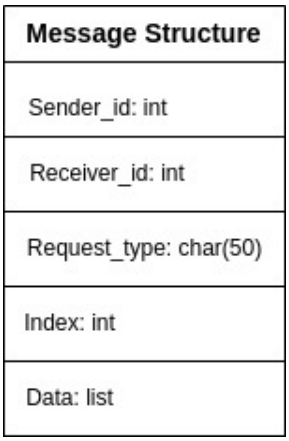


Figure 4.12: The structure of the communication message

Message dispatch

ZMQ was employed for message transmission, and MessagePack was utilized for message serialization. To dispatch a message, following the completion of all the aforementioned fields, the message undergoes serialization through MessagePack to transform it into bytes. Subsequently, it is transmitted via the ZMQ socket to the designated recipient node. The recipient's address comprises the IP address, represented by the name of the hosted container, and the port, which is set to 5555 for communication between group nodes and 4555 for communication between leader nodes, during the initial configuration. The fig below we can see that process.

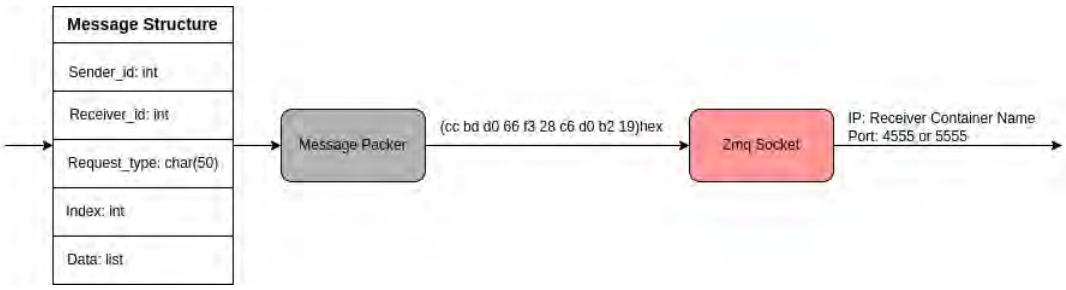


Figure 4.13: process for dispatching a message

Message receipt

Corresponding to the reception of the message, the process proceeds by capturing the message via the ZMQ socket. The message is then passed through MessagePack, performing the conversion from bytes to a Python dictionary. This transformation allows the node to decode the message and extract the relevant information necessary for the seamless continuation

of the experiment.

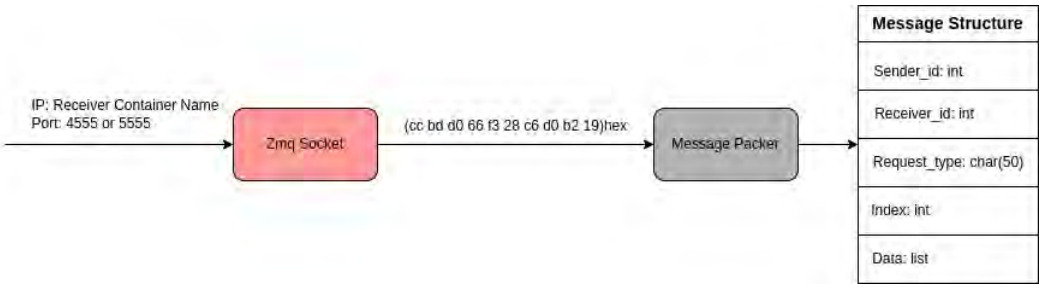


Figure 4.14: process for receiving a message

4.3.7 Communication with the databases

Connection to MySQL

For communication with MySQL, each node utilized the mysql.connector library. The host is specified as the name of the database container, which, in our instance, is mysqldb_statistics, and the port is set to 33060, as defined in the corresponding .yml file. This configuration enables each node to seamlessly store all pertinent information related to the experiment in the MySQL database.

Connection to MongoDB

To facilitate communication between each node and MongoDB, the pymongo library was employed. In configuring the connection, the host is specified as the name of the database container, specifically, mongodb_datasets, with port 27017 as defined in the corresponding .yml file. This setup enables each node to efficiently retrieve a portion of the dataset, crucial for the accurate execution of the experiment. The utilization of pymongo streamlines the interaction between the Python application and the MongoDB database, providing a robust mechanism for data retrieval within the experimental context.

4.3.8 Diversities between FDA and HDA

The distinctions between HDA (Hybrid Docker Architecture) and FDA (Full Docker Architecture) implementations are as follows: HDA comprises only two Docker containers, in contrast to FDA's four, which encompass the previously described databases. Additionally, the remaining processes operate locally on the computer. Despite sharing identical code, the

sole difference lies in the epochal address assigned to each node. Since they execute locally, nodes utilize the ”localhost” IP address, with ports differing based on each node’s ID specifically, port 5555 + node_id. Another notable difference is that the controller, for container creation, opts for multiprocessing over the Docker SDK library, creating processes instead. These disparities constitute the entirety of variances in the implementations. Notably, the controller conveniently accommodates both local and Docker execution by accepting an ”is_docker” parameter, ensuring ease of use in either format.

4.3.9 PDFL simulator project file structure

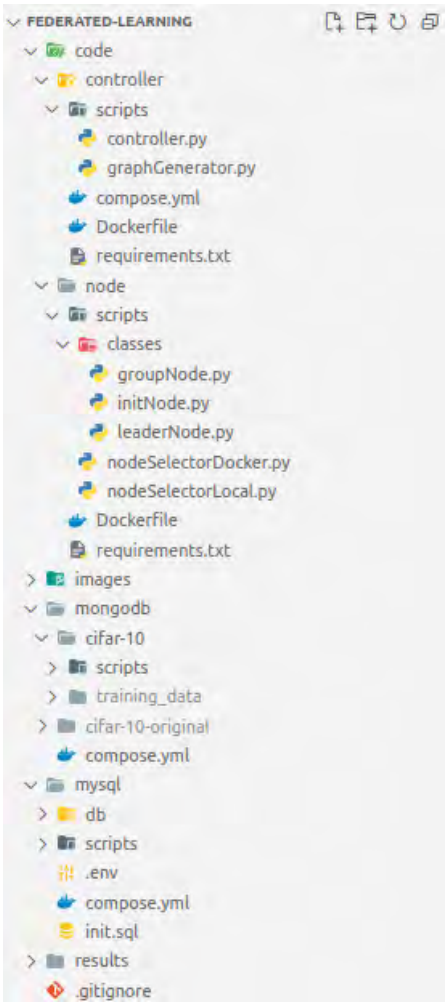


Figure 4.15: PDFL simulator project file structure

Here, we examine the source code of the simulator, organized into four distinct folders, each corresponding to a different container. The first set includes the controller’s folder, housing both the source code and Docker runtime files, a structure similarly mirrored in the

node's folder. Two additional folders are dedicated to the databases, providing all necessary files. Lastly, we encounter a results folder. Following the conclusion of an experiment, the controller generates an Excel file in this directory, encapsulating comprehensive information gleaned from the experimental run. This modular organization facilitates a structured and comprehensible architecture for the simulator's source code and associated resources.

4.4 How it operates

Its operation is simple. The user only needs to start the controller container, providing the necessary inputs as previously described. Once the controller is operational, it goes through three stages, each of which will be discussed in detail below.

4.4.1 Initialization

At startup, the controller triggers the graphGenerator class to construct a graph based on the user's specifications. Next, it initializes the tcp_listener thread, facilitating communication with other containers. Finally, the controller creates as many node containers as there are nodes in the graph, using the docker SDK library, providing them with the appropriate parameters for proper initialization. Upon successful completion of these tasks, the initial stage of the program is completed, ensuring the correct initialization of all containers.

4.4.2 Simulation of the Graph

All the containers have been created. The Controller sends information about the topology of the graph to each node separately. So that each node knows which node to communicate with during the execution of the experiment. Once they receive this message they store the useful information in the message. They also communicate with the mongoDB database, so they have the dataset of their neural network. Subsequently, a directive is sent from the Controller to initiate the graph simulation. Upon receiving this signal, nodes commence training their neural networks, engaging in message exchange. The performance metrics are then saved into the MySQL database to facilitate the seamless operation of federated learning. In figure below we can see the simulation. In figure below, we can see the containers during simulation.

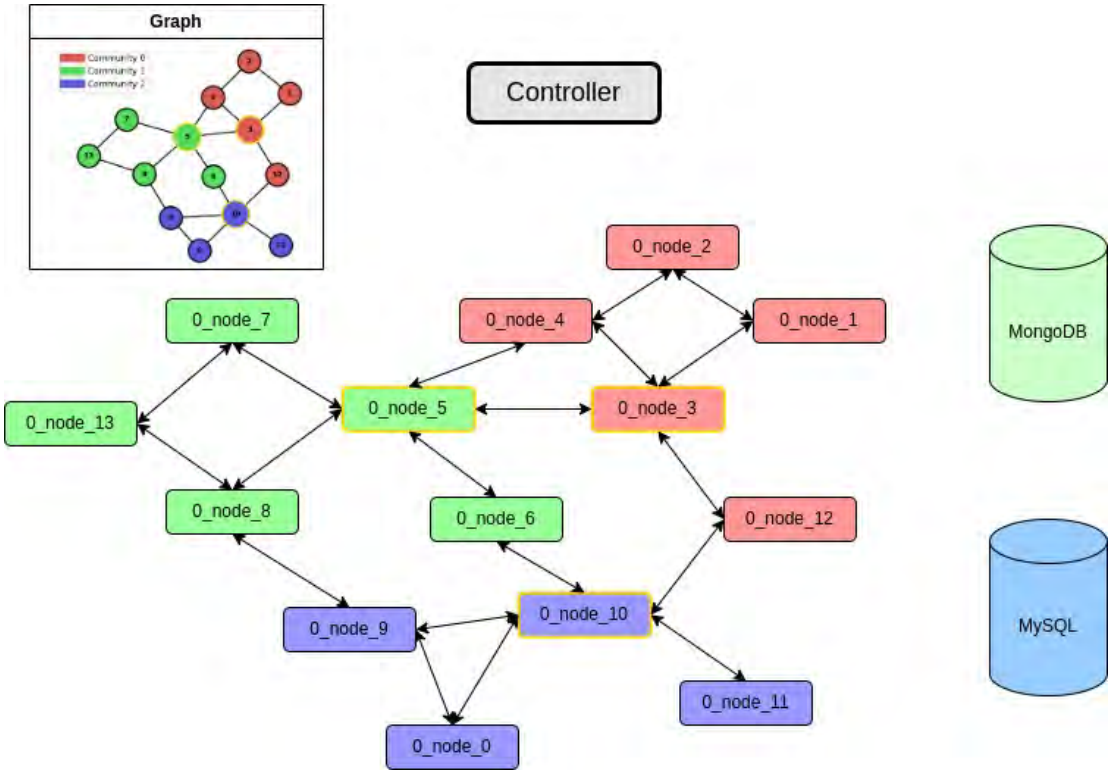


Figure 4.16: Appearance of simulator during simulation

4.4.3 Termination

As we have mentioned above, training nodes take as a definition the total training repetitions of their neural network. Once they reach this limit, they inform the controller that they have reached the end of their training. Thus, these nodes close all threads except their tcp_listener, so that the remaining nodes that have not finished can continue the experiment. Finally, after the controller receives the required number of termination messages, it informs the nodes to terminate. After the node containers have stopped working the controller deletes them permanently, as they have completed their purpose.The final task of the controller is to pull all the data of the particular experiment, recorded by the nodes and create an excel file with all the information of the experiment. So in the end the user can see how the experiment has progressed.

Chapter 5

Evaluation

In this section, we will start by presenting the dataset used and delve into the neural network architecture employed for each node. Then, we will examine the results obtained from our experiments. These experiments are testing the algorithms and new methods previously discussed in the context of the project we developed. Our goal is to validate the proper functionality of this program and to extract insights from the experimental results.

5.1 Data set details

The selection of data plays a significant role in the efficiency of our proposed model. Since we are working on federated learning which is mainly used in IoT devices (e.g. smartphones, wearable devices, autonomous vehicles) we decided to use image classification data for prediction. Particularly, we used a real data set obtained by Kaggle [33], the CIFAR-10 dataset [34] is a widely used benchmark in the field of computer vision and machine learning. Developed by the Canadian Institute for Advanced Research (CIFAR), CIFAR-10 consists of 60,000 32x32 color images in 10 different classes, with each class representing a distinct object or scene category. Ten thousand testing pictures and fifty thousand training pictures make up the dataset. The ten classes in CIFAR-10 are: Airplane, Automobile, Bird, Cat, Deer, Dog, Frog, Horse, Ship, Truck.

CIFAR-10 serves as a standard benchmark for evaluating the performance of machine learning algorithms, especially in the context of image classification tasks. Researchers and practitioners often use it to assess and compare the effectiveness of different models and techniques in image recognition. The relatively small size of the images and the diverse set

of classes make CIFAR-10 a challenging but manageable dataset for experimentation and development of new algorithms.

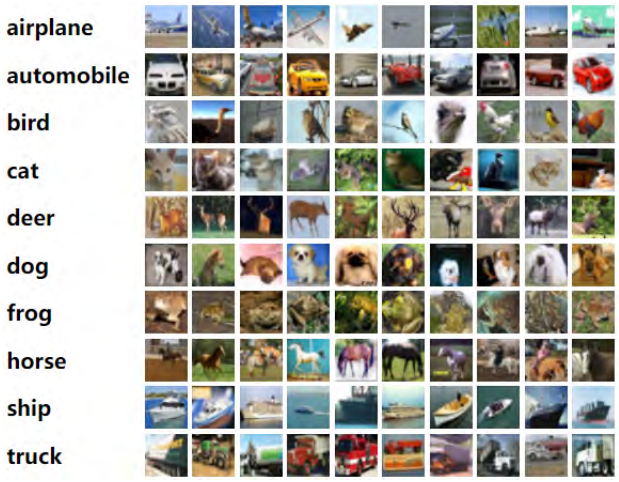


Figure 5.1: CIFAR10 Dataset

5.2 Neural network Architecture

In the pursuit of leveraging Federated Learning (FL) techniques for decentralized and privacy-preserving model training, a Convolutional Neural Network (CNN) architecture was carefully crafted for the specific demands of the CIFAR-10 dataset. The goal was to enable collaborative training across distributed devices while maintaining the integrity of individual data.

The architecture embraces principles of federated learning by incorporating mechanisms to secure privacy and enable collaborative model updates. The structure begins with convolutional layers designed to capture intricate patterns and features inherent in the CIFAR-10 images. Each convolutional layer is followed by batch normalization, ensuring stable and consistent activations across devices.

To facilitate decentralized learning, dropout layers are strategically inserted at various stages of the network. These dropout layers serve the dual purpose of preventing overfitting and encouraging the network to learn robust features that generalize well across different devices in a federated setting.

Max-pooling layers are interspersed within the architecture to downsample spatial dimensions, allowing the network to focus on essential features while minimizing the transmission of redundant information. The depth of the network is gradually increased, empowering

the model to learn hierarchical representations crucial for recognizing diverse objects in the CIFAR-10 dataset.

The final layers of the architecture consist of densely connected layers, culminating in a softmax-activated output layer. This configuration enables the model to output class probabilities, suitable for the multi-class nature of the CIFAR-10 dataset.

This neural network architecture forms the foundation for federated learning experiments. In the context of federated learning, this architecture is trained collaboratively across devices, with each device only transmitting model updates rather than raw data. The decentralized and privacy-preserving nature of federated learning aligns with the ethos of ensuring user privacy while enabling the collective improvement of the model.

For model compilation in federated learning, Stochastic Gradient Descent (SGD) with a learning rate of 0.001 and a momentum term of 0.9 is employed. The categorical cross-entropy loss function is utilized to quantify the discrepancy between predicted and actual class probabilities.

This neural network serves as a crucial component in the exploration of federated learning on the CIFAR-10 dataset, exemplifying the intersection of deep learning and decentralized collaboration for enhanced model training.

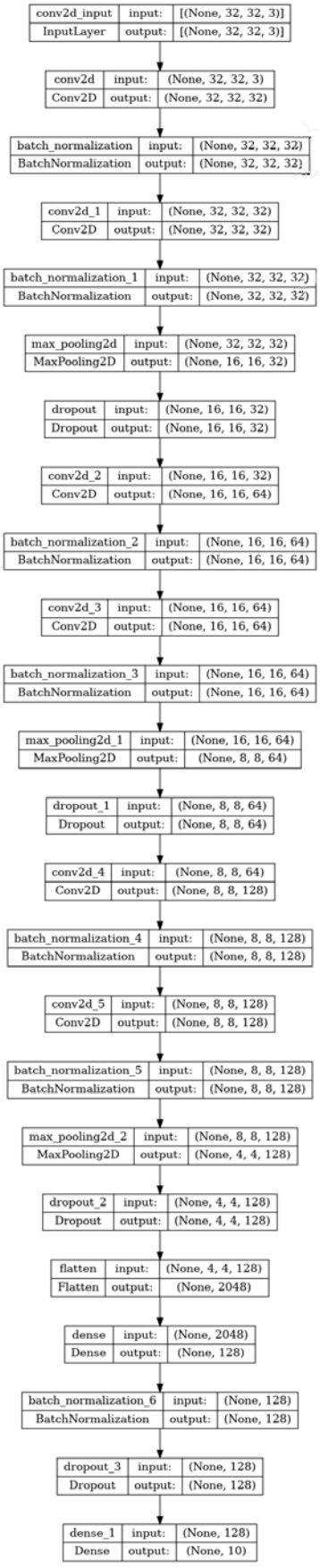


Figure 5.2: Neural Network Architecture

5.3 Results

5.3.1 Overview

A compact network consisting of six nodes, three in each subnetwork, was established to compute the final results. This design was primarily necessitated by RAM constraints on the computer used for the experiments, creating a more manageable architecture. The accompanying Figure illustrates the network configuration.

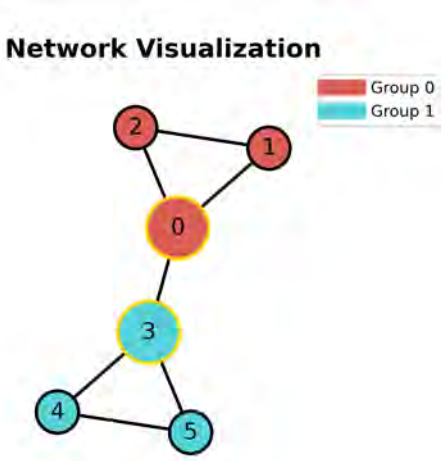
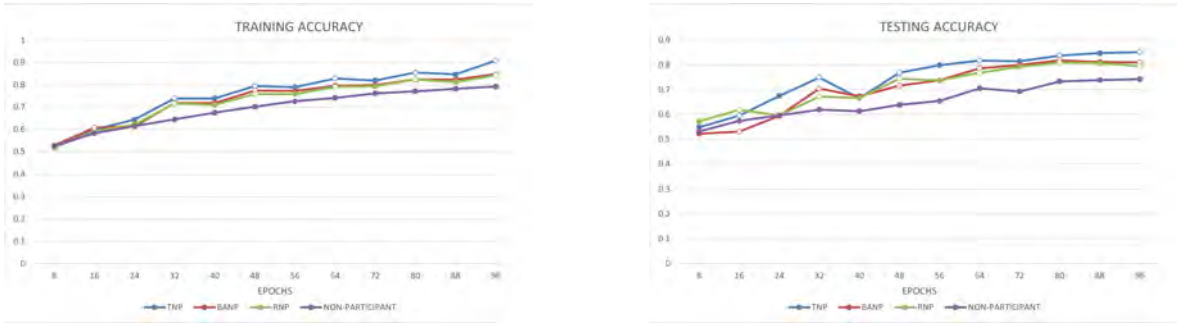


Figure 5.3: Network Visualization

Three experiments were conducted, each corresponding to a node participation algorithm: Total Participation (TNP), Random Node Participation (RNP), and Best Accuracy Node Participation (BANP), with a participation rate of 70%. That means that four of the six nodes participate in FL. Additionally, a standalone experiment was executed for a single node's non-participation in federated learning, facilitating a comparative analysis of participation effectiveness.

Each experiment involved fifteen thousand training images and ten thousand evaluation images for every node. Data was communicated to the network at sixteen-epoch intervals, resulting in 96 training rounds and six rounds of data transmission.

5.3.2 Accuracy Comparison



(α′) Training accuracy

(β′) Testing accuracy

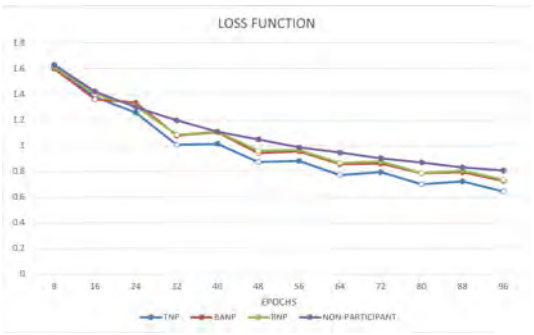
Figure 5.4: Accuracy comparison

In Figure 5.4, it is evident that participation in federated learning consistently leads to higher accuracy compared to a non-participating node (indicated by the purple line). This observation holds true across all three experiments, where accuracy consistently outperformed the non-participating scenario. Precision calculations (represented by the TNP, BANP, and RNP lines) were derived by computing the mean of each node at a specific time, contributing to these noteworthy results.

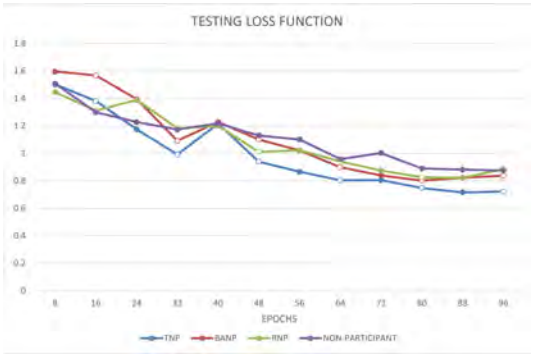
Examining the white dots on the Figure, it becomes apparent that these dots mark instances when nodes receive new weights. Consequently, immediate performance improvement is limited in the subsequent time step. However, a sustained use of the same weights reveals a substantial increase in precision, surpassing the non-participant line by a significant margin.

Concerning the results of TNP, BANP, and RNP, it is rational to observe that TNP yields the best outcomes. This is attributed to the smaller scale of the experiment's network, ensuring that nodes in other experiments, where participation decreases, do not absorb the entire dataset of the neural network.

5.3.3 Loss Function Comparison



(α′) Training Loss function



(β′) Testing Loss function

Figure 5.5: Loss function comparison

As seen overhead in Figure 5.5, it becomes apparent that engagement in federated learning consistently results in lower loss values than a non-participating node, as indicated by the downward trend of the purple line. This trend holds across all three experiments, with lower loss consistently observed in participation scenarios. To calculate the loss (reflected in the TNP, BANP, and RNP lines), we computed the mean loss of each node at a given time, contributing to these findings.

Notably, the white dots on the graph signify instances when nodes receive new weights. At these points, immediate reduction in loss is limited in the subsequent time step. However, persisting with the same weights reveals a substantial decrease in loss, surpassing the non-participant line by a considerable margin.

Regarding the TNP, BANP, and RNP results, it is logical to observe that TNP exhibits the most favorable outcomes. This can be attributed to the smaller scale of the experiment's network, ensuring that nodes in other experiments, where participation decreases, do not dominate the entire loss computation of the neural network.

5.3.4 Total Messages Comparison

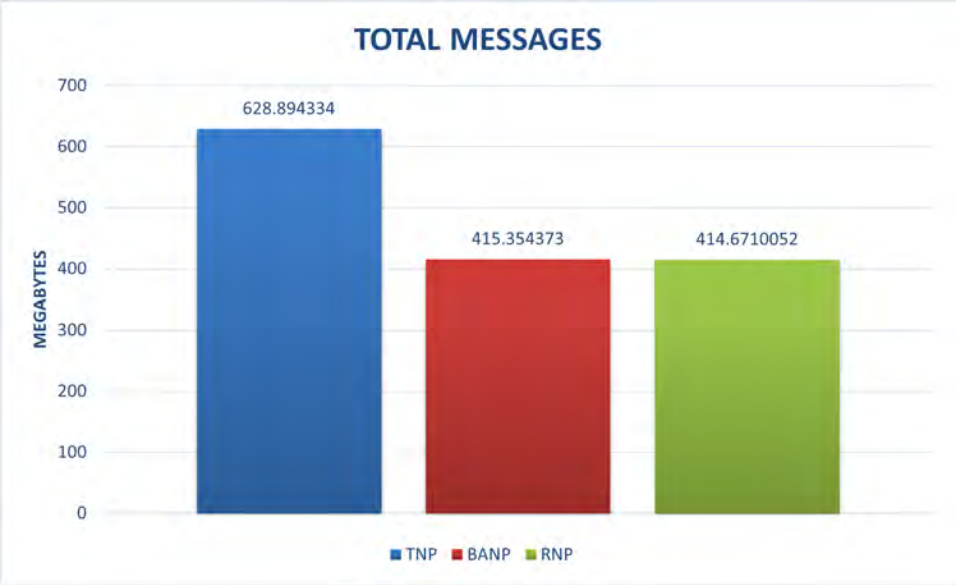


Figure 5.6: Total Messages Comparison

Examining Figure 5.6, depicting the total messages transmitted during the experiments in gigabytes, it's crucial to note that these figures encompass both the controller's messages with the nodes and the dominant node-to-node communication. The messages exchanged between the controller and the nodes are minimal compared to inter-node communication.

This outcome is innately logical, given that the TNP algorithm utilizes all six nodes in the network. In comparison, the BANP and RNP algorithms employ only four out of the six nodes, resulting in a 70% participation rate. Notably, BANP registers a slightly higher message count attributed to the network leaders retaining information about discarded nodes after the initial neural training for accuracy assessment. In contrast, RNP discards nodes before the start of training, thereby reducing the overall message communication.

Chapter 6

Conclusions

6.1 Summary and Conclusion

In this thesis, we introduced an innovative approach to decentralized deep learning models tailored for devices interconnected through wireless networks. Our primary objectives were twofold: addressing privacy preservation challenges and curbing communication costs. To achieve these aims, we designed a model devoid of a central server for coordinating aggregation. Instead, devices are organized into clusters, with a designated cluster head responsible for aggregating model updates within each cluster. Subsequently, these cluster heads, acting as representatives of their respective clusters, engage in communication and parameter exchange to execute the overarching aggregation process.

The empirical foundation of our work rests on real-world data extracted from the cifar10 dataset. We devised a series of participation policies for nodes in federated learning, including full participation, random participation, and best-accurancy participation. These policies were subjected to experimentation.

Finally, we developed an application, which is housed in a Docker environment and leverages multiple Python libraries to facilitate the exploration of different techniques and efficiently simulate the experiment. This application not only allows users to test different methodologies, but also provides a user-friendly interface for storing and reviewing the results of each experiment. This ensures that users maintain full control and gain valuable insights into the results of their efforts.

6.2 Future work

Future work for the thesis could be with the addition of a web application where the user could create their own graph, but also the results produced by each node to see them in real time and better judge the results. With all his experiments available for viewing at any time.

Bibliography

[1] Latif U. Khan Walid Saad. Zhu Han Ekram Hossain and Choong Seon Hong. Federated learning for internet of things: Recent advances, taxonomy, and open challenges. *IEEE Communications Surveys and Tutorials*, 23(21075505):1759 – 1799, June 2021.

[2] Yanming Guo Yu Liu Ard Oerlemans Songyang Lao Song Wu amd Michael S. Lew. Deep learning for visual understanding: A review. *Neurocomputing*, 187:27–48, April 2016.

[3] What is reinforcement learning? `https://aws.amazon.com/what-is/reinforcement-learning/`.

[4] Supervised machine learning. `https://www.datacamp.com/blog/supervised-machine-learning/`.

[5] Types of neural networks and definition of neural network. `https://www.mygreatlearning.com/blog/types-of-neural-networks/`.

[6] Siddharth Sharma Simone Sharma and Anidhya Athaiya. Activation functions in neural networks. *International Journal of Engineering Applied Sciences and Technology*, 4:310–316, April 2020.

[7] Introduction to loss functions. `https://www.datarobot.com/blog/introduction-to-loss-functions/`.

[8] Enrique Tomás Martínez Beltrán Mario Quiles Pérez Pedro Miguel Sánchez Sánchez Sergio López Bernal Gérôme Bovet Manuel Gil Pérez Gregorio Martínez Pérez and Alberto Huertas Celdrán. Decentralized federated learning: Fundamentals, state of the art, frameworks, trends, and challenges. *IEEE Communications Surveys and Tutorials*, 25:2983 – 3013, Sep. 2023.

[9] Benefits of federated learning explained. https://octaipipe.ai/benefits-of-federated-learning-explained/.

[10] Li Li Yuxi Fan Mike Tse Kuo-Yi Lin. A review of applications in federated learning. *Computers and Industrial Engineering*, 149(106854), Nov. 2020.

[11] Rui Xu and D. Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(8422347):645 – 678, May 2005.

[12] Stephen P. Borgatti. Centrality and network flow. *Social Networks*, 27:55–71, Jan. 2005.

[13] Watts–strogatz model. https://en.wikipedia.org/wiki/Watts%E2%80%93Strogatz_model.

[14] Louvain clustering algorithm. https://towardsdatascience.com/louvain-algorithm-93fde589f58c.

[15] Closeness centrality. https://www.geeksforgeeks.org/closeness-centrality-centrality-measure/.

[16] Jed Mills Jia Hu and Geyong Min. Communication-efficient federated learning for wireless edge intelligence in iot. *IEEE Internet of Things Journal*, 7:5986 – 5994, July 2020.

[17] Docker. https://www.docker.com/.

[18] Docker-virtual-networks. https://spacelift.io/blog/docker-networking.

[19] Mysql database. https://www.mysql.com/.

[20] Mongodb database. https://www.mongodb.com/.

[21] Docker-sdk. https://docker-py.readthedocs.io/en/stable/containers.html.

[22] Zeromq. https://zeromq.org/.

[23] Messagepack. https://msgpack.org/.

[24] Numpy. https://numpy.org/.

[25] Pandas. https://pandas.pydata.org/.

[26] Tensorflow. https://www.tensorflow.org/.

[27] Keras. https://keras.io/.

[28] Kerastuner. https://keras.io/keras_tuner/.

[29] Multithreading-in-python. https://www.geeksforgeeks.org/multithreading-python-set-1/.

[30] Threadpoolexecutor. https://docs.python.org/3/library/concurrent.futures.html.

[31] Networkx. https://networkx.org/.

[32] Matplotlib. https://matplotlib.org/.

[33] Kaggle. https://www.kaggle.com/.

[34] cifar10. https://en.wikipedia.org/wiki/CIFAR-10.