POLITECNICO DI TORINO

Master's Degree in Communications Engineering

Master's Degree Thesis

Machine Learning for 5G/6G

Supervisor

Prof. Carla Fabiana CHIASSERINI

Candidate

Mahyar ONSORI

April 2025

Abstract

Distributed learning schemes have emerged as promising alternatives to traditional centralized machine learning, especially in 5G environments where data is generated at the edge.

This thesis investigates the impact of non-Independent and Identically Distributed (non-IID) data on the performance of various learning schemes, federated and decentralized learning, using the UCI-HAR dataset as a proof-of-concept, since it is a light real-world dataset suitable for simulating non-IID scenarios. We developed a scalable pipeline which is able to generate different non-IID distributions, We develop a modular pipeline capable of generating multiple non-IID distributions, simulating diverse network topologies, and evaluating model performance under varied hyperparameters and distance metrics.

Our results show that when the heterogeneity in the data increases, centralized aggregation methods fail to converge rapidly, and over a certain epoch, they are suboptimal solutions. In contrast, decentralized approaches perform good under non-IID distribution of the data.

As we said, in this paper, we developed two main distributed learning schemes, however, some upgraded schemes were planned and can be developed in the future works. One of them is an extension of decentralized learning, named reduced decentralized learning, which tries to carefully remove the network links with different metrics, and can reduce communication overhead, latency, and energy consumption, with minimum decrease in the performance.

We finish the article by reviewing the obtained results, and pointing the main limitations and challenges, and possible enhancements for future works. The most important enhancements are working with a larger dataset, adding more metrics for measuring the performance of the learning schemes, and trying to reduce the gap between our simulation environment and real-world scenarios.

Keywords: Distributed Learning, 5G, Non-IID Data, Decentralized Learning, Federated Learning

Acknowledgements

First of all, I want to thank Prof. Carla Fabiana Chiasserini, for all her support through this way. I appreciate her advices and help to pass all the challenges that I faced.

I want to thank my colleagues, Dr. Francesco Malandrino, and Dr. Carlos Barroso Fernandez. I know without them, it was impossible to complete this journey. It was an honor for me to work with you.

I am grateful to HPC Polito, who allowed me to use their resources during the experiments of this thesis.

I want to thank all my classmates in Polito. Maybe our path will never cross again in the life, but I will remember all good moments that we had together until last second of my life. Polito will have an invaluable place at my heart.

I want to thank my friends who helped me to settle in Turin. Mohammadhossein, Mahdi, Iman, Mohammad, Ali, Arash and Mohammad, Good memories that we had here will always bring smile to my face, and I was lucky to have you guys. I am very grateful to my friends in Iran, Ali, Hadi, Erfan, Seyed, Mohammad, Amin, and Sadegh, I wish you were here, but thanks for your support.

Finally, I want to say a big thank you and a big I love you to my family, I know I would not achieve anything in my life without them. My mother's love, my father's support, my sister's energy, and Yasaman's encouragement were my strength to beat all obstacles. I hope that I can compensate a bit of their support, although I know words are not capable to fulfil my emotions.

"One day, you'll leave this world behind. So live a life you will remember"
Nicholas Furlong

Table of Contents

List of Tables

List of Figures

Acronyms

AI

artificial intelligence

ML

machine learning

FL

federated learning

ETL

e-tree learning

DL

decentralized learning

rDL

reduced decentralized learning

SVD

Singular Value Decomposition

KMA

K-Means Algorithm

IoT

Internet of Things

IID

Independent and identically distributed

HAR	Human Activity Recognition
SDN	Software Defined Networking

Chapter 1

Introduction

1.1 Background and Motivation

5G is the fifth generation of mobile networks. In comparison with its predecessor, 4G, 5G offers some important features: higher data rates, ultra-low latency, greater reliability and availability, and increased network capacity[1].

Several techniques have been used to achieve these features. Spectrum efficiency, massive MIMO, beamforming, and SDN-based network architecture are the most significant techniques that have been used[2]. By using higher frequency bands, 5G provides larger bandwidth; thus, it affects network capacity and availability, in addition to higher data rates. Beamforming allows us to focus on the signals of the users to improve signal quality. By adopting an SDN-based network architecture, network slicing is enabled[3], providing a situation where different requirements of different users can be met. This feature is useful when network requirements vary from one user to another, or even for a user over a period of time.

Due to the aforementioned features of 5G, it has been deployed in many applications, most of which also incorporate AI. One of the most famous use cases are smart cars and the autonomous driving. By expanding 5G networks, it is possible to have vehicles, pedestrians, and infrastructure connected to each other. Vehicles can exchange different types of data, such as traffic information or collision warnings, with each other, and communicate with the infrastructure to receive information and spread it throughout the network[3][4][5]. Another use case is the integration of 5G into industry. Due to the high network capacity and low latency, 5G networks can meet the requirements of smart factories. Now, with an increased number of smart devices, as well as moving entities such as robots and drones in factories, 5G can help in operating, monitoring, and maintaining the industrial processes[2].

This topic will be covered in Chapter 2; however, it is worth noting that traditional ML solutions were based on centralized data collection and training, which is impractical in 5G use cases for several reasons. The first concern is privacy. Storing all data in a single location is risky, because if this information is leaked, a vast number of individuals can be affected by the exposure of their data. Also, as the size of the data increases, ensuring privacy becomes more challenging, and individuals have less control over how their data is used. In addition to concerns about user privacy, data security can be challenging in a centralized scenario. Since malicious parties can attack this centralized database and steal sensitive data, strong security algorithms must be implemented, which are complex and costly. Furthermore, centralized solutions have problems with the network, and more specifically, bandwidth. Transmitting all data from different nodes to a centralized database requires significant bandwidth and can lead to high latencies or even network interruptions, which are not tolerated by real-time services. Therefore, using distributed learning methods is vital. In these distributed learning methods, data is processed locally so that it is not centralized in a database nor distributed across the network. Instead, other information, such as the weights in training ML models, is exchanged between the nodes.

1.2 Problem Statement

The distribution of data plays a key role in designing and training machine learning models. In statistics, data is said to be independent and identically distributed (IID) if and only if all random variables are mutually independent and share the same probability distribution.

This assumption is important in classical machine learning solutions, all of which are based on it. As we previously mentioned, traditional machine learning solutions were centralized; a server trained the model and sent the weights back to the nodes. Under the assumption of IID data, this solution is effective, as it assumes that the data is homogeneous and that sharing the same weights across all nodes does not create a problem.

However, in real-world scenarios such as 5G/6G use cases, it is not possible to assume that the data is IID, since the dataset for each node is captured by different sensors or under different circumstances. In a 5G network, the sensors used for the collection of data can differ, the network graph may change over time (affecting the distribution), and the nodes' characteristics may also differ. Therefore, in real-world problems we mostly deal with non-IID data. Although traditional ML solutions can still perform acceptably, it is preferable to work with non-IID data and design models accordingly.

In the federated learning scheme that will be described later, working with non-IID data can be challenging. Since federated learning still uses a central node to collect local updates and update the global model sent back to the nodes, non-IID data causes slower convergence and poorer global model accuracy due to the intrinsic heterogeneity of the data among the nodes. Because the data is diverse, a global model that fits all the data and performs well on every task is rarely practical[6]. This confirms the importance of investigating distributed and decentralized learning schemes.

1.3 Research Objectives

1.3.1 Distributions

In this paper, we designed and developed tools to generate different types of data distributions to mimic real-world non-IID scenarios. This will be discussed further in Chapter 3. The random distribution is the first distribution implemented by our tool. In this distribution, data is randomly divided into nearly equal partitions, and class balance in each portion can be guaranteed if the data is split randomly. This distribution simulates scenarios in which data is unstructured across nodes, yet balanced in terms of size.

The next distribution is the class-specific distribution, where we assign classes to nodes selectively to simulate data imbalance, which in turn translates into heterogeneity. This scenario closely resembles real-world cases in which data is diverse in terms of size and distribution, such as user-specific data in mobile networks. The final distribution implemented was the beta distribution, which controls data imbalance by adjusting the beta parameter of the beta probability distribution. This scenario is less common in real-world settings than the previous distributions, but it can be used to test the performance and robustness of the schemes against data heterogeneity, since the beta parameter allows for systematic variation of the imbalance.

1.3.2 Learning Schemes

In this research, different learning schemes were designed and deployed. The first scheme was the federated learning scheme, in which there is a server node and many device nodes. Each node trains its model independently, and the server aggregates all local models to generate a global model, which is then tested. The E-Tree learning scheme is a cluster-based variant of the federated learning scheme. In this scheme, nodes are grouped into clusters; each cluster aggregates its model locally, and a global model is created from the cluster centers. Clustering can be done randomly or using a clustering algorithm. Local updates are aggregated at the cluster centers, which then synchronize with a global server. The next scheme is decentralized learning, in which there is no server or central node; instead, nodes communicate only with their neighbors in the graph, and aggregation occurs locally, where first, the neighbors for each node are determined, and after each node trains its model, the models are aggregated locally.

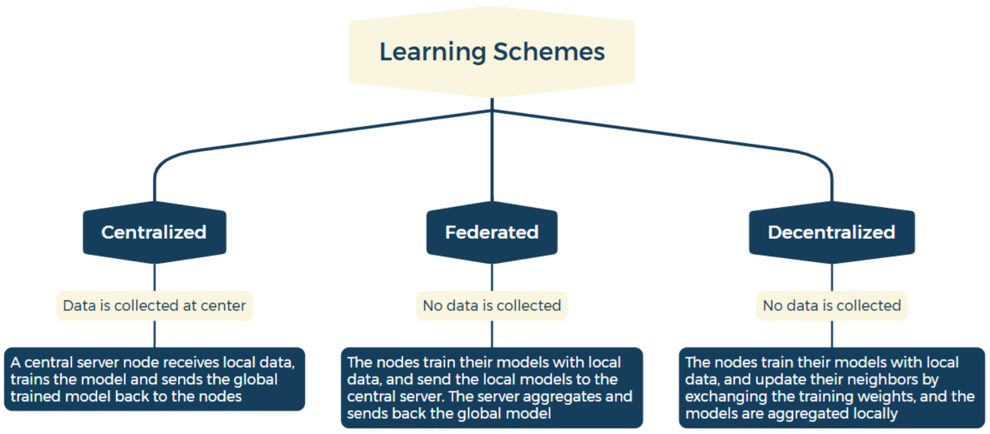
Learning Schemes
Centralized
Federated
Decentralized
Data is collected at center
No data is collected
No data is collected
A central server node receives local data, trains the model and sends the global trained model back to the nodes
The nodes train their models with local data, and send the local models to the central server. The server aggregates and sends back the global model
The nodes train their models with local data, and update their neighbors by exchanging the training weights, and the models are aggregated locally

Figure 1.1: Overview of different learning schemes.

1.3.3 Heterogeneity

In this section, different distance metrics that were used to quantify the impact of heterogeneity are briefly introduced. The Frobenius norm is the first distance metric used; it is useful for comparing feature subspaces, such as the SVD components of a dataset. It is important in situations where datasets differ in their feature representations, as it captures structural similarity. The second metric is the L2 distance, which quantifies feature differences by measuring the Euclidean distance between normalized distributions. The final metric is the Wasserstein distance, which primarily evaluates distribution alignment. These metrics play a crucial role in quantifying dataset heterogeneity.

1.4 Contributions

In this paper, our main goal was to design and develop a pipeline to explore different learning schemes, which were mainly focused on distributed learning schemes. We wanted to compare the performance of these schemes under different conditions,

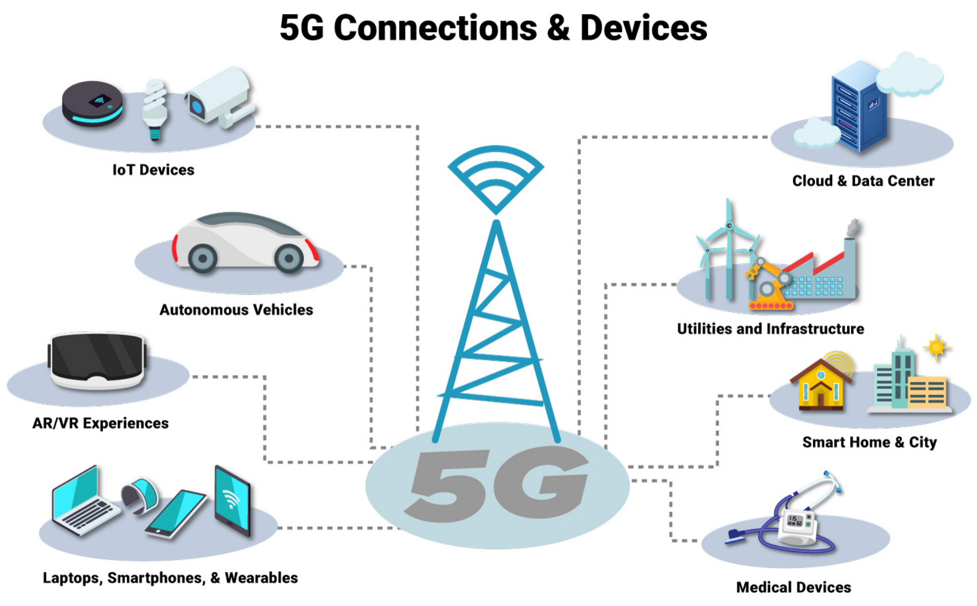

Figure 1.2: 5G devices generating data.

especially under different data distributions, and also to study the performance of the different schemes on the same data distribution. To reach our goals, we had to develop different tools. The first tool was a dataset loading module, where we developed the functions to properly load and preprocess the dataset.

We developed a tool for generating the graph needed for simulating the network, and another tool for clustering algorithms used during the pipeline. More importantly, we created a module for generating different data distributions to mimic IID and non-IID scenarios. We designed the structure of the learning schemes and the training pipeline, which served as the module integrating all other parts.

These tools provide an opportunity to investigate the performance of different learning schemes for 5G networks. We aimed to take into account the effects of various factors that influence system performance or play a key role in real-world applications, in order to create a strong baseline for future research in distributed learning deployment in 5G networks, by developing a modular system consisting of distribution generation, graph generation, a training pipeline, and results analysis.

1.5 Thesis Outline

This paper is structured as follows: In Chapter 2, the related literature is reviewed, and important topics such as the integration of machine learning in 5G networks, an introduction to federated and decentralized learning, and non-IID data distributions are discussed to provide a common understanding of the paper's key aspects. In Chapter 3, we describe our contributions to the research objectives. The pipeline designed to implement the objectives, the data distribution mechanisms for generating the required experimental distributions, and the deployed training schemes are explained. In Chapter 4, we first discuss the experimental setup and implementation details, and then present and discuss the obtained results, concluding with the key findings.

Chapter 2

Literature Review

2.1 Machine Learning in 5G/6G

The fast evolution of cellular networks, from 4G to 5G and 6G, has created both an opportunity and a necessity to deploy ML solutions to address networking challenges. In modern cellular networks, ML algorithms are being used for different tasks, such as dynamic spectrum allocation, resource management, interference mitigation, and quality-of-service (QoS) optimization. By feeding historical records of data usage, incoming data from sensors, and real-time status of the network, ML solutions can provide predictive maintenance, traffic forecasting, and adaptive control mechanisms that can enhance both network efficiency and user experience [7].

A trending development in the ML field is the increasing need for privacy-aware learning. Traditional ML approaches, which were centralized, were based on aggregating raw data at a central server, which leads to concerns about privacy and data ownership. To solve these problems, distributed learning algorithms are being developed, which can be seen as on-device learning. In these learning schemes, edge devices train local models with their own data, and only share their model updates or weights, rather than the data itself. This solution can help address privacy concerns, as well as reduce latency and network load, which are key factors for a real-time network [8] [9].

The shift towards distributed learning not only addresses privacy but also helps devices rapidly adapt to ongoing changes, such as fluctuations in network loads, which contributes to a more robust network infrastructure. Also, the computational power at edge devices is increasing, and they are increasingly capable of supporting complex ML training [10].

6G networks are being designed with principles that have ML as an intrinsic feature. Many tasks like traffic prediction, routing, and security in 6G could be handled by distributed AI models, achieving ultra-reliable and adaptive network performance in real time. Early 6G studies highlight use-cases such as intelligent resource allocation, network self-healing, and advanced security analytics driven by machine learning in simulation and testbeds. While 6G is not yet deployed, these studies indicate a trend where future networks will have ML algorithms integrated as core components for decision-making and automation [11].

In summary, the application of ML in modern cellular networks is affecting network management. Distributed learning schemes are being developed to secure user data privacy, as well as to enable more effective network management tasks, to reduce latency and energy consumption and also to increase real-time adaptability.

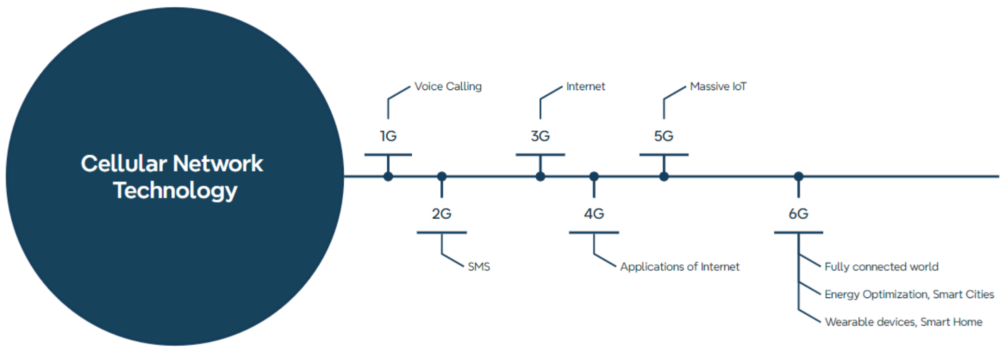

Figure 2.1: Evolution of mobile network technologies.

2.2 Federated and Decentralized Learning

2.2.1 Federated Learning

Federated learning (FL) is one of the main distributed learning schemes which allows multiple devices to collaboratively learn a model while keeping their sensitive training data localized. In the FL architecture, a central server named orchestrator manages the training process and the distribution of the weights. Each device trains its local model on its data, sends its model updates to the server, and the server aggregates the local models and produces a new global model, which is again sent to the users for further local training by each device. The process is repeated until the model converges [12] [13].

Although traditional FL, which is based on a server-client model, is a good baseline for collaborative learning, it also has some potential drawbacks. The main problem of this scenario is its vulnerability to a single point of failure. One of the ways to solve the problems of the classical FL method is to use a hierarchical FL, which creates groups or clusters from the nodes, and then follows an approach similar to the FL approach, with the difference of adding a local aggregation step at the cluster centers. Each node trains its local model on the local data, updates the cluster center about the model, the cluster center aggregates its cluster members' models and sends the aggregated model to the central node, which aggregates all the models and sends back the global model to the cluster centers, which then send it back to their members. This hierarchical approach can lead to a reduction in communication overhead, and overcome the challenge of data diversity, especially in cases where devices are heterogeneous [14] [15].

2.2.2 Decentralized Learning

Another type of distributed learning is decentralized or peer-to-peer learning, in which there is no central node in the network. Instead, each node trains its model locally and shares its model updates with each other[16] [17]. By doing these updates, model parameters converge throughout the network. The decentralized approach is more scalable and robust against a single point of failure, but as mentioned in [18], network topology plays a key role in this approach.

An important note in the decentralized learning approach, as mentioned in [19], is the problem of lazy nodes. Lazy clients are the nodes that use the trained models of their neighbors, add artificial noise to make a change, and distribute it in the network. This is one of the most important challenges of decentralized learning that needs to be addressed.

By developing distributed learning methods, some concerns about data privacy are solved, and by using the increasing computational power of edge devices, distributed learning solutions will be more widely deployed.

2.3 Non-IID Data Distributions

In probability theory, a data distribution is referred to as independent and identically distributed (IID) if all random variables in that distribution possess those features. In many scientific fields, including machine learning, the main and classical assumption about the data was that it is IID. Although this assumption simplifies problem solving, it also introduces significant challenges.

The IID assumption only deals with the subset of the problem space where it holds, and leads to a gap in existing solutions, making them biased towards IID data. However, real-world complex problems usually involve non-IID data, which leads to problems for classical ML solutions [20].

Due to the heterogeneity in the nodes' data in distributed learning, especially in FL, data cannot be considered IID, and this non-IID nature is one of the key challenges for FL. Different nodes have different data distributions, which can result in a degradation in global model performance [21].

According to [22], there are some solutions to address the problem of non-IID data in FL. The solutions can be divided into data-based, model-based, algorithm-based, and framework-based. Data-based approaches mostly focus on compensating for the non-IID nature of the data, by sharing it or enhancing it. Model-based solutions try to fix the problem by modifying the optimization techniques or changing the way that the global model is aggregated.

2.4 Gap Analysis

In the different studies on distributed learning schemes, there is a gap in the comparison of FL and DL under different heterogeneous conditions. In [17], they compared using a few different distributions and a fixed network graph. In [23], they compared FL and DL with different network configurations, but not with multiple data distributions. Our work not only compares different distributed schemes with a variety of options for altering configurations and parameters, but also provides a reliable baseline for future work to compare even the advanced or upgraded versions of learning schemes.

As federated learning (with a central server) and fully decentralized learning are two different paradigms, researchers have investigated their comparative performance and limitations. A key question has been: Which approach performs better, and under what conditions? Recent comparative studies provide valuable insights. Hegedűs et al. conducted an empirical comparison of classic federated learning (FL) and a gossip-based decentralized learning on various tasks [24]. Interestingly, they found that a well-designed gossip protocol (fully decentralized, no server) can match the accuracy of federated learning in many cases, despite being purely peer-to-peer. This result suggests that decentralized schemes are a viable alternative, achieving similar model quality without a central coordinator. However, the study also noted efficiency trade-offs: gossip learning required more communication rounds to converge (since information spreads gradually), and careful tuning (e.g. controlling message traffic and handling client churn) was needed to prevent lags[24].

Another recent work by Sun et al. takes a theoretical perspective on centralized FL (CFL) vs. decentralized FL (DFL) [25]. They proved that, under general non-convex objectives, CFL tends to generalize better than DFL in terms of model accuracy. In other words, having a central server aggregate updates yields a model with lower generalization error compared to fully decentralized training, especially as the network scales. Their analysis also highlights a crucial limitation of decentralized learning: the network topology plays a significant role. They show that DFL can suffer a performance collapse if the peer-to-peer network topology is not well-connected as the number of clients grows. Simply put, if the decentralized network is poorly structured (e.g. sparse or bottlenecked), the training may diverge as more nodes join. This points to an important gap – DFL requires careful topology design (or additional mechanisms) to scale reliably, an area needing further research. On the flip side, centralized FL has its own gaps: the central server can become a bottleneck or single point of failure/trust, and partial participation (having only a fraction of clients communicate each round) often yields better results than trying to include everyone every round [25].

The comparisons above indicate that neither approach is strictly superior in all aspects; each has limitations that open avenues for research. One gap is network topology optimization for decentralized learning, e.g., how to form communication graphs that ensure fast convergence and resilience to dropouts. Another open issue is the trade-off between communication cost and accuracy: decentralized methods can reduce server load but might incur more total communication among peers; more studies are needed to quantify this in real-world network settings (especially wireless or P2P networks). Hybrid architectures are also being explored: for instance, multi-tier federated learning (with clusters of nodes gossiping locally and a server syncing the clusters) or using blockchain-based consensus to add trust in decentralized updates. Furthermore, theoretical bounds and convergence guarantees for decentralized algorithms in realistic non-IID scenarios are still an active research area. Overall, the gap analysis suggests that while federated and decentralized learning each hold promise, practical deployments might end up combining elements of both (to balance reliability and scalability). Continued research is needed to address issues like client/network heterogeneity, fault tolerance, and performance tuning in different topologies.

Chapter 3

Methodology

3.1 Overall Experimental Pipeline

For this paper, we had to first choose a dataset. We opted to work with the UCI-HAR dataset, since it is a small dataset that is suitable for proof-of-concept applications, and also it has diverse data that can resemble the diversity in a real-world 5G application dataset. Then we loaded the dataset and preprocess it to make it ready for further steps.

In the next phase, non-IID distributions are generated, and the dataset is divided into different sections. As we explained before, in a real-world scenario it is very likely that the data is non-IID, due to the heterogeneity in circumstances, sensors, and users' behaviors. Then, we select a topology for our network to simulate the real network graph between the nodes in a 5G scenario. For our paper, we used a random graph and a mesh graph; however, it is possible to generate different topologies such as full, lollipop, or star topologies. We chose the random topology, since in real cases some nodes may join or leave the network, and the graph will appear random. However, the results for the mesh topology are also presented in Chapter 4.

Finally, a scheme was selected from the developed schemes, and the model is trained on the provided dataset with the desired parameters. There are some hyperparameters that the user can choose, and we can compare their effects by executing different training runs. Lastly, using a variety of plots and metrics, the performance of the model is evaluated.

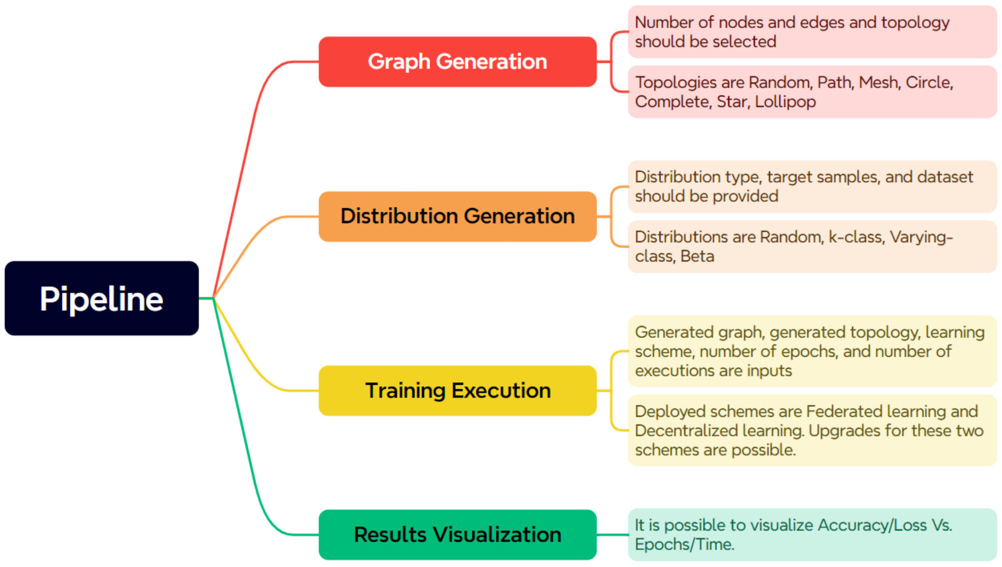
Pipeline
Graph Generation
Number of nodes and edges and topology should be selected
Topologies are Random, Path, Mesh, Circle, Complete, Star, Lollipop
Distribution Generation
Distribution type, target samples, and dataset should be provided
Distributions are Random, k-class, Varying-class, Beta
Training Execution
Generated graph, generated topology, learning scheme, number of epochs, and number of executions are inputs
Deployed schemes are Federated learning and Decentralized learning. Upgrades for these two schemes are possible.
Results Visualization
It is possible to visualize Accuracy/Loss Vs. Epochs/Time.

Figure 3.1: A conceptual figure of the experiments pipeline.

3.2 Data Distribution Mechanisms

Federated learning operates in decentralized environments where data and conditions are highly variable. We define this variety as heterogeneity. In essence, heterogeneity in FL captures the differences between clients in terms of data distributions, computational capabilities, availability, and network constraints. Understanding and addressing these heterogeneous conditions is crucial for designing robust and fair federated learning systems. There are different types of heterogeneity, such as data heterogeneity, network heterogeneity, and behavioral heterogeneity. For our purposes, data heterogeneity is the most important. As we said earlier, each client may observe data specific to its context or user base, resulting in skewed or imbalanced local distributions, which leads to non-IID data. Data heterogeneity highly affects the design and evaluation of federated learning systems. By effectively addressing heterogeneity, the main challenge of balancing collaboration and privacy can be achieved.

3.2.1 Distribution Methods

In the script for generating the distributions, we generated different scenarios for the dataset. The first scenario was a *random distribution*, where the dataset was split into N classes of the same size. The next scenario that we implemented was a *class selective distribution*, where we can assign certain classes to the nodes. In our implementation, it is possible to assign a specific number of classes to each node (such as one class per node, two classes per node), or even to assign the data more precisely by imposing conditions—for example, ensuring that a specific node does not receive certain classes. This approach can resemble situations where nodes are specialized in specific tasks.

Finally, we implemented a *beta distribution* to simulate skewed datasets and create controlled imbalance situations. As discussed in Chapter 1, the beta distribution provides a systematic way to simulate skewness and imbalance in the data. The probability density function (PDF) for this distribution is given by

$$f(p;\alpha,\beta) = \frac{1}{B(\alpha,\beta)}, p^{\alpha-1}(1-p)^{\beta-1}, \quad 0 < p < 1, \tag{3.1}$$

where $B(\alpha,\beta)$ is the beta function. If we set $\alpha = 0.5$ and $\beta = 0.5$, the distribution will be of exact "U" shape. However, if we change the values, the distribution still has a U-shaped form but becomes skewed. For example, if $\alpha = 0.2$ and $\beta = 0.8$, the distribution is U-shaped but is skewed heavily towards 0 compared to 1.

Table 3.1: Summary of Different Distributions

Distribution Name	**Description**
Random	Representative of IID distribution in experiments. Each node selects an almost equal number of samples from all classes.
k-class	One of the non-IID distributions deployed. In this distribution, each node selects all its samples from exactly k classes of the total classes. As k decreases, the distribution gets far from IID.
Varying-class	Another non-IID distribution. In this distribution, there is no constraint on the data for nodes. They may contain data from all the classes, or some specific classes. There is no obligation that the number of classes among different nodes stays the same.
Beta	This non-IID distribution represents the skewness in the dataset. It has two parameters named alpha and beta. If $\alpha = \beta = 0.5$, the distribution is perfectly "U" shaped, but in other cases, data is skewed towards one of the ends of the range.

3.2.2 Heterogeneity Metrics

We also measured the distance between the distributions to quantify heterogeneity. To do so, we considered the L2 (Euclidean), Wasserstein, and Frobenius distance metrics (their formulas are given below). Eventually, we decided to continue with the L2 distance metric.

Frobenius Norm

The Frobenius norm is a matrix norm that calculates the root of the sum of the squared differences between the corresponding matrix elements. In the context of this study, it is applied to compare feature subspaces, such as Singular Value Decomposition (SVD) components, across datasets. This metric helps identify structural dissimilarities or similarities in datasets, particularly when feature representations differ significantly among nodes. It is especially useful for examining heterogeneity at the feature level.

Formula:

$$|A - B|F = \sqrt{\sum i = 1^m \sum_{j=1}^{n} (a_{ij} - b_{ij})^2} \tag{3.2}$$

where A and B are matrices representing dataset features.

L2/Euclidean Distance

The L2 distance measures the straight-line distance between two points in a multidimensional space. In this study, it quantifies the differences in normalized feature distributions between datasets. Its simplicity and efficiency make it a practical metric for large-scale evaluations of data heterogeneity. This metric is particularly effective for detecting feature-level disparities in non-IID settings.

Formula:

$$|x - y|2 = \sqrt{\sum i = 1^n (x_i - y_i)^2} \tag{3.3}$$

where x and y are feature vectors.

Wasserstein/Earth Mover's Distance

The Wasserstein Distance evaluates the cost of transporting one probability distribution to another, making it ideal for assessing distributional alignment or misalignment across datasets. Unlike the L2 distance, which focuses on feature disparities, the Wasserstein distance captures the "shape" and "alignment" of distributions. This metric is particularly valuable in cases of imbalanced or skewed data distributions.

Formula (1D case):

$$W(p, q) = \int_{-\infty}^{\infty} |P(x) - Q(x)| \, , dx \tag{3.4}$$

where $P(x)$ and $Q(x)$ are the cumulative distribution functions.

These metrics collectively provide a comprehensive understanding of dataset heterogeneity. The Frobenius Norm measures structural alignment, the L2 Distance quantifies feature disparities, and the Wasserstein Distance evaluates distribution alignment. Together, they help analyze how heterogeneity affects model performance, guiding the development of learning algorithms that are robust to non-IID data.

3.3 Network Topology and Clustering

3.3.1 Graph Generation

Another script we developed was related to simulating different network topologies. By deploying the NetworkX library in Python, we were able to generate different topologies to mimic various requirements. In our tool, it is possible to generate a variety of topologies, such as complete, mesh, star, unconnected, random, circle, and path. Also, by modifying some of these pre-built graph types, we can create customized graphs, which are useful for testing purposes. After a graph is selected, random weights are given to the edges of the graph. This assignment of weights is beneficial for simulating real-world scenarios where the connections between different nodes are not the same, and it also helps to evaluate the performance of the system under uneven conditions.

Here are some examples of the possible graphs generated by our script.

- Path Graph

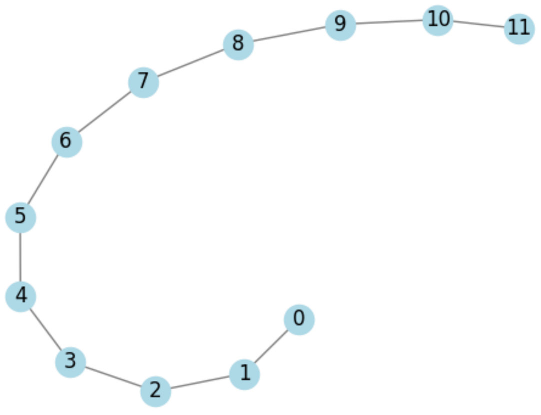

Figure 3.2: An example of a path graph with N = 12 nodes.

- Complete Graph

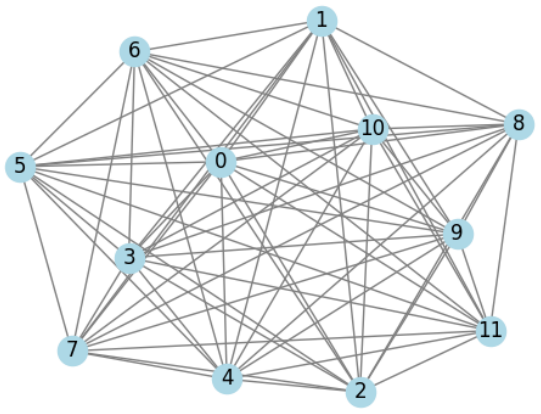

Figure 3.3: An example of a complete graph with N = 12 nodes.

- Mesh Graph

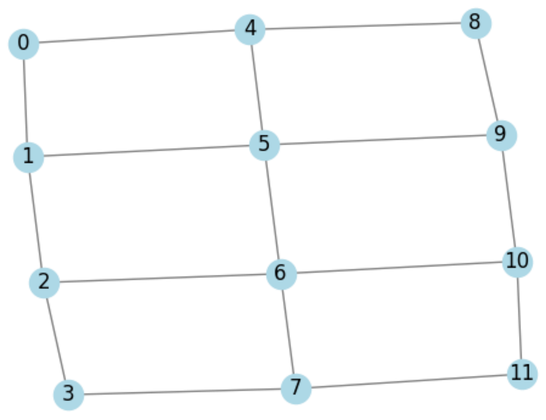

Figure 3.4: An example of a mesh graph with N = 12 nodes.

- Random Graph

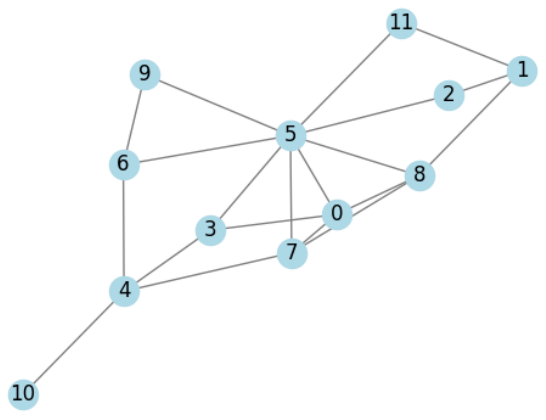

Figure 3.5: An example of a random graph with N = 12 nodes and e = 20 edges.

3.3.2 Clustering Algorithms

As discussed earlier, one of the methods used in this paper is ETL. For the implementation of ETL, we had to implement different clustering methods. The first method is to divide nodes into N clusters of the same size. This is useful in scenarios that require balance among the nodes.

The next clustering algorithm that was deployed was random clustering, which randomly assigns nodes to clusters, whose centers are also picked randomly. This method can act as a baseline for comparing other clustering techniques. A more advanced technique that was used is K-means. The K-means algorithm starts by receiving the number of clusters, then some random points are set as the initial centers of the clusters. The rest of the nodes are assigned to the clusters based on their distance; after assignment, new centers are calculated, and then nodes are reassigned based on their proximity to the new centers. The algorithm continues until there is no change in the clustering in consecutive iterations.

In an upgrade to the K-means algorithm, we introduced a second version with a small change in the early stages. The difference is that it computes the distances beforehand, so it is more efficient and less time-consuming. The last clustering method used, named KMA, is again an upgrade to the second version of K-means. In KMA, there is an extra constraint: a node is only added to a cluster if it does not push the cluster's average accuracy beyond δ from the global average accuracy.

3.4 Federated and Decentralized Schemes

In this paper, three main schemes were designed to be used in the experiments.

3.4.1 Federated Learning

The first one is federated learning (FL). In this scheme, one of the nodes is selected as the central node, which plays a key role in FL. In FL, each node trains its model independently using its data, and sends the model iteratively to the central node.

The central node aggregates all the local models to build a global model and sends the global model back to the nodes. It also validates the global model by tracking its validation loss. If the validation loss converges and stays almost constant, which translates to an L shape, or diverges after convergence, which translates to a U shape, the training loop for all the nodes is stopped. This scheme relies heavily on a consistent connection between the nodes and the central node.

3.4.2 E-Tree Learning

The second scheme that was deployed was E-Tree learning (ETL). This scheme is close to FL, but this time nodes are divided into different clusters, and there is an additional step of aggregating the local models in the clusters. The first part is like FL, as each node trains its model locally. N nodes are picked as the initial centers of the clusters, and then the closest points to these nodes are assigned to that cluster.

After each node trains its own model, the models are aggregated locally in each cluster by the central node of the cluster. Then, the global model is created from the cluster centers. Again, this procedure is monitored by the behavior of validation loss, as in FL. ETL is highly scalable due to its hierarchical structure and cluster-based implementation. It can be seen as a more distributed version of FL, because some local aggregation near the edges is done, and then the final aggregation happens at the center.

3.4.3 Decentralized Learning

The third scheme that was implemented in this paper was decentralized learning (DL). This scheme removes the need for a central node—whether a single central node as in FL or local central nodes as in ETL—thereby making DL less prone to single-point failures, which is especially important for 5G/6G applications. In DL, there are peer-to-peer connections between the nodes, and they only communicate with their neighbors in the graph. Aggregation still occurs, but it is done locally among the neighbors. Also, validation is performed by each node in the same way as in FL and ETL.

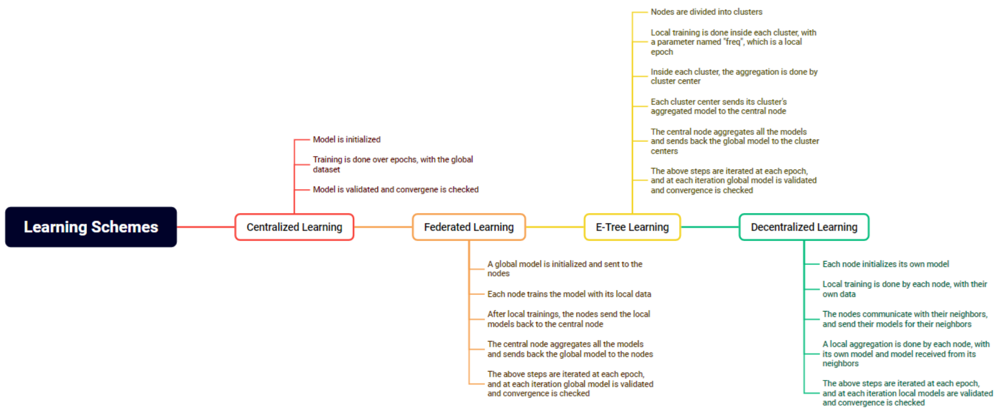
Learning Schemes
Centralized Learning
Model is initialized
Training is done over epochs, with the global dataset
Model is validated and convergene is checked
Federated Learning
A global model is initialized and sent to the nodes
Each node trains the model with its local data
After local trainings, the nodes send the local models back to the central node
The central node aggregates all the models and sends back the global model to the nodes
The above steps are iterated at each epoch, and at each iteration global model is validated and convergence is checked
E-Tree Learning
Nodes are divided into clusters
Local training is done inside each cluster, with a parameter named "freq", which is a local epoch
Inside each cluster, the aggregation is done by cluster center
Each cluster center sends its cluster's aggregated model to the central node
The central node aggregates all the models and sends back the global model to the cluster centers
The above steps are iterated at each epoch, and at each iteration global model is validated and convergence is checked
Decentralized Learning
Each node initializes its own model
Local training is done by each node, with their own data
The nodes communicate with their neighbors, and send their models for their neighbors
A local aggregation is done by each node, with its own model and model received from its neighbors
The above steps are iterated at each epoch, and at each iteration local models are validated and convergence is checked

Figure 3.6: A summary of the steps for different schemes.

3.5 Implementation Flow

In this paper, we organized the pipeline into a series of scripts, each dedicated to a specific phase of implementation. In the following figure, an illustration of the pipeline is shown.

Data Preparation

The preparation starts with the HARfunctions.py script, which, as the name suggests, is designed to preprocess the HAR dataset; however, we generalized its functions as much as possible to be compatible with future experiments. Using the generatedistribution.py script, we split the dataset into different subsets using the provided distributions that can mimic IID and non-IID scenarios.

Graph Generation

Another step before executing the training is to create and store the desired network topology for the simulation by running generateGraph.py with specified settings.

Training Execution

In the main training step of the pipeline, trainModel.py is executed. While there are some mutual parameters between different learning schemes, they also have exclusive parameters. The procedure is to load the pickle files output by previous steps, and call the functions from trainingSchemes.py to run local training, model aggregation, and decentralized updates if needed. Final performance metrics are then stored in a file to be later processed.

Result Visualization

After finishing the training, using plotRes.py, we can generate different accuracy or loss curves, box plots, or correlation heatmaps. Also, for some metric comparisons, we can run readCSVfiles.py to analyze CSV logs generated, if needed.

3.6 Reproducibility Notes

For implementing this paper, we opted to work with the Python programming language as it is a pioneering language in the ML field, and it eases the implementation of the training schemes as well as network simulation. For the latter, we used the NetworkX library, where we can generate different graphs based on our requirements and test different network topologies. This library simulates the communication networks required for distributed learning. For designing the models and the pipeline, the PyTorch library was selected, as it provides a variety of options for proper implementation of the intended schemes. In the final phases, we used pandas to handle and process the datasets. Also, for illustration purposes, matplotlib and seaborn were used. The versions for each used library can be seen in Table 1. The flow for training a scheme is to generate the proper graph, generate the distribution, and feed them as inputs to trainmodel.py, and then with plotres.py it is possible to draw a variety of plots comparing different parameters.

Chapter 4

Implementation

4.1 Experimental Setup

To run the experiments and implement the described methods in this thesis, the following environments were used:

Hardware Environment

- **CPU:** Intel Core i7-7500U, 2-core, 2.7 GHz, and Intel Core i5-10210U, 4-core, 2.11 GHz.
- **GPU:** NVIDIA GeForce 940MX, 4 GB GDDR5.
- **RAM:** 20 GB DDR4, 3200 MHz and 16 GB DDR4, 3200 MHz.
- **Storage:** 1 TB NVMe SSD and 256 GB NVMe SSD.

Software Environment

- **Operating System:** Windows 10 Pro and Windows 11 Pro.
- **Programming Language:** Python 3.10 (simulations) and Python 3.12 (data analysis).
- **Frameworks and Libraries:**
  - PyTorch: 2.2.1, for implementing and training machine learning models.
  - NetworkX: 3.2.1, for generating and analyzing graph-based topologies.
  - Matplotlib: 3.8.4 and Seaborn: 0.13.2, for data visualization.
  - Pandas: 2.2.1, for data manipulation and processing.
  - NumPy: 1.26.4, for numerical operations and computations.

– Scipy: 1.13.0, for numerical operations and computations.

Development Tools

- **IDE:** Visual Studio Code, Jupyter Notebook, Spyder.
- **Version Control:** GitHub for managing the codebase.

The above-mentioned hardware and software setup provided a proper environment for running the experiments efficiently.

Table 4.1: Experimental Setup and Training Parameters

Category	**Component**	**Specification**
Hardware Environment	CPU	Intel Core i7-7500U (2-core, 2.7 GHz) and Intel Core i5-10210U (4-core, 2.11 GHz)
	GPU	NVIDIA GeForce 940MX, 4 GB GDDR5
	RAM	20 GB DDR4, 3200 MHz and 16 GB DDR4, 3200 MHz
	Storage	1 TB NVMe SSD and 256 GB NVMe SSD
Software Environment	Operating System	Windows 10 Pro and Windows 11 Pro
	Programming Language	Python 3.10 (simulations) and Python 3.12 (data analysis)
	PyTorch	2.2.1
	NetworkX	3.2.1
	Matplotlib, Seaborn	Matplotlib 3.8.4 and Seaborn 0.13.2
	Pandas	2.2.1
	NumPy, Scipy	NumPy 1.26.4 and Scipy 1.13.0
Development Tools	IDE	Visual Studio Code, Jupyter Notebook, Spyder
	Version Control	GitHub
Training Parameters	Learning Rate	(0.001)
	Number of Epochs	(100)

In trainingSchemes.py, we introduced an update to the DL scheme, named rDL, which removes the edges based on a logic (random, minimum V-distance, etc.).

This proposed scheme tries to keep the performance as high as possible but reduces latency, network load, and energy consumption.

In generateGraph.py, we generated different network graphs based on the NetworkX library in Python, but adjusted to meet our requirements. We tried to set up a baseline for the future, where more complex graphs can be tested for specific simulation scenarios.

In the next chapter, the results are shown and discussed. For testing the effect of different metrics and parameters, many experiments were conducted, and parameters and metrics were evaluated. It is possible to generate scenarios with different network graphs and data distributions for all of the deployed schemes.

4.2 Implementation Flow

In this paper, we tried to organize the pipeline into a series of scripts, where each one is dedicated to a specific phase of implementation. In the following figure, an illustration of the pipeline is shown.

Data Preparation

The preparation starts with the HARfunctions.py script, which, as the name suggests, is designed to preprocess the HAR dataset, but we tried to generalize its functions as much as possible to be compatible with future experiments. Using the generatedistribution.py script, we split the dataset into different subsets using the provided distributions that can mimic IID and non-IID scenarios.

Graph Generation

Another step before executing the training is to create and store the desired network topology for the simulation by running generateGraph.py with specified settings.

Training Execution

In the main training step of the pipeline, trainModel.py is executed. While there are some mutual parameters between different learning schemes, they have exclusive parameters. The procedure is to load the pickle files output by previous steps and call the functions from trainingSchemes.py to run local training, model aggregation, and decentralized updates if needed. Final performance metrics are then stored in a file for later processing.

Result Visualization

After finishing the training, using plotRes.py, we can generate different accuracy or loss curves, box plots, or correlation heatmaps. Also, for some metric comparisons, we can run readCSVfiles.py to analyze CSV logs generated, if needed.

4.3 Reproducibility Notes

For implementing this paper, we opted to work with the Python programming language as it is a pioneering language in the ML field, and it eases the implementation of the training schemes as well as network simulation. For the latter, we used the NetworkX library, with which we can generate different graphs based on our requirements and test different network topologies. For designing the models and the pipeline, the PyTorch library was selected, as it provides a variety of options for proper implementation of the intended schemes. In the final phases, we used pandas to handle and process the datasets. Also, for illustration purposes, matplotlib and seaborn were used. The versions for each used library can be seen in Table 1. The flow for training a scheme is to generate the proper graph, generate the distribution, and feed them as inputs to trainmodel.py, and then, with plotres.py, it is possible to draw a variety of plots comparing different parameters.

Chapter 5

Results

5.1 Results and Analysis

In this section, we present the results obtained from the experiments. Note that the number of experiments run for this thesis was very high, and for simplicity and convenience, only the most useful plots are shown.

Also, it should be noted that our pipeline is capable of generating all metric plots (such as accuracy, precision, recall, and F1 score), but we decided to continue with accuracy, since it is the most common metric in distributed learning performance analysis.

5.1.1 Simulated Scenarios

As mentioned above, to investigate the effect of different parameters and metrics, many experiments were carried out, and some metrics were favored over others. For the experiments presented in this chapter, we used FL and DL as the two main learning schemes. In addition, two different network graphs with 7 nodes were used to show the effect of network graph and node connectivity on the performance of the learning schemes.

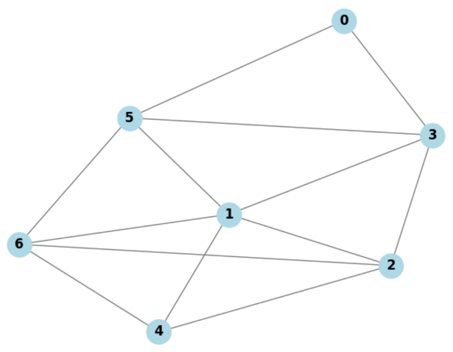

Figure 5.1: Random graph used in these experiments.

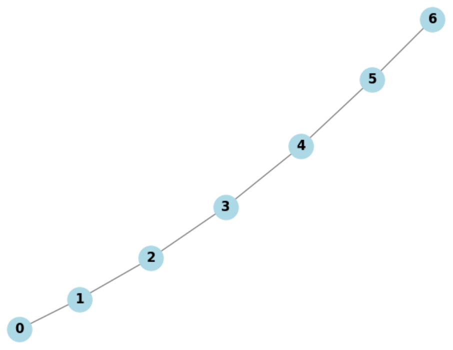

Figure 5.2: Mesh graph used in these experiments.

For the distribution of the data, four scenarios were chosen. The first distribution is a random distribution, which, as mentioned in the previous chapters, resembles the IID data distribution. This distribution is mostly used to highlight the difference between IID and non-IID data and to show how heterogeneity affects the performance of the models, as conventional ML solutions may not overcome the challenge of non-IID data.

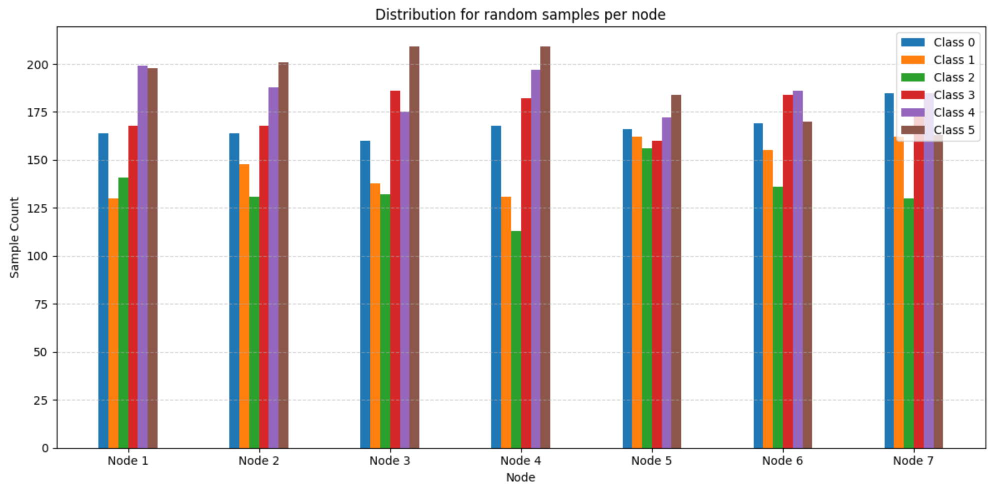

Figure 5.3: Random distribution used in these experiments.

Another distribution that was deployed is one in which each node can have data without any constraint. To explain better, in contrast to the previous scenario—in which each node had data from all six classes in the dataset—in this distribution each node may have all its data from one class or from specific classes, and may not represent all classes. This distribution has the highest heterogeneity and resembles real-world examples of 5G networks, in which data across the nodes may vary significantly. We refer to this distribution as the "Varying Class" distribution in this chapter.

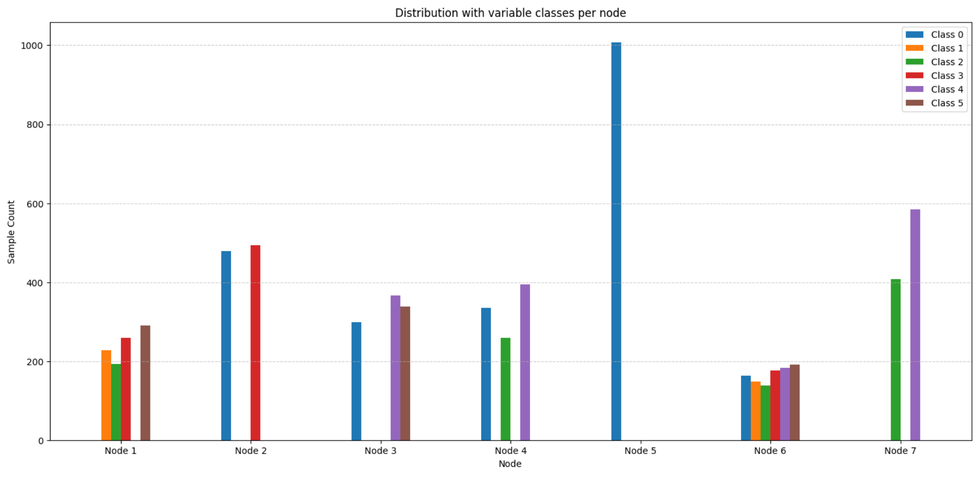

Figure 5.4: Varying class distribution used in these experiments.

The last two distributions that we tested are class-specific distributions, in which each node selects exactly k classes out of 6 classes in the dataset. The portion of the data is selected randomly, and it is possible that one of the k classes has the majority of the data, or is a minor class, or that the data is split almost equally among the k classes. In this set of experiments, we used three distributions with k = 2 and k = 4.

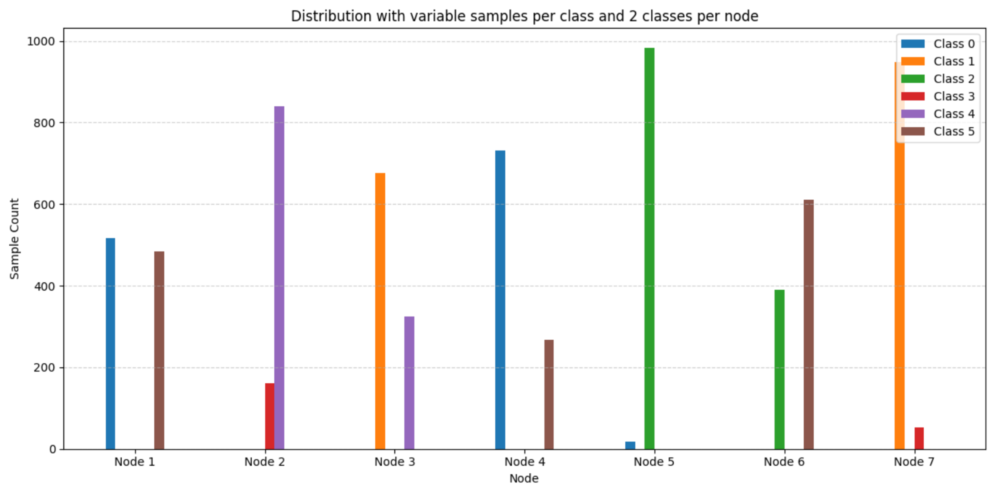

Figure 5.5: Specific class distributions used in these experiments with k = 2.

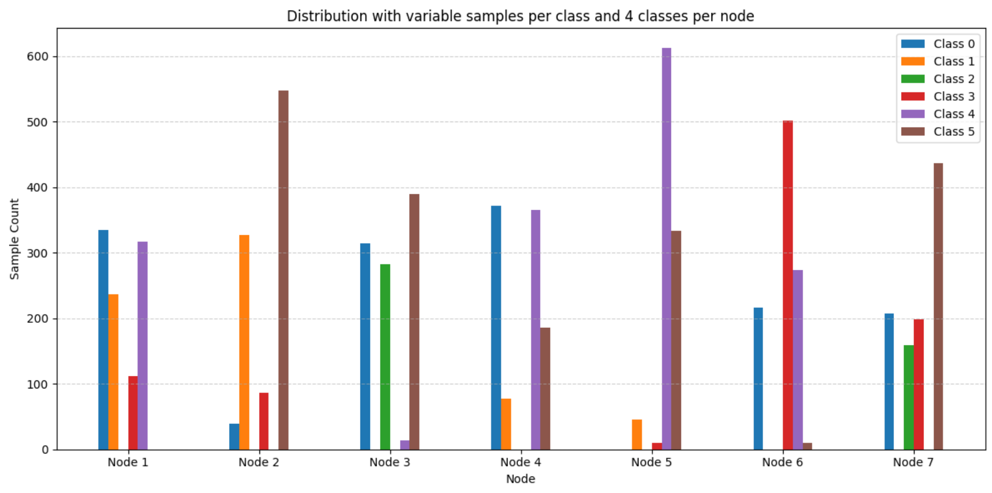

Figure 5.6: Specific class distributions used in these experiments with k = 4.

The above review was necessary to understand the experimental environment, and in the next section we present the results and discuss them.

5.2 Experiments Results and Discussion

In this section, we present the results for different combinations of the hyperparameters, and we also show comparison plots between them.

5.2.1 Learning Schemes Performance

In this subsection, the results for comparing the performance of our two main learning schemes are shown. Therefore, the rest of the parameters remain the same to isolate the effect of the learning scheme.

- Random Graph, Two-Class Distribution

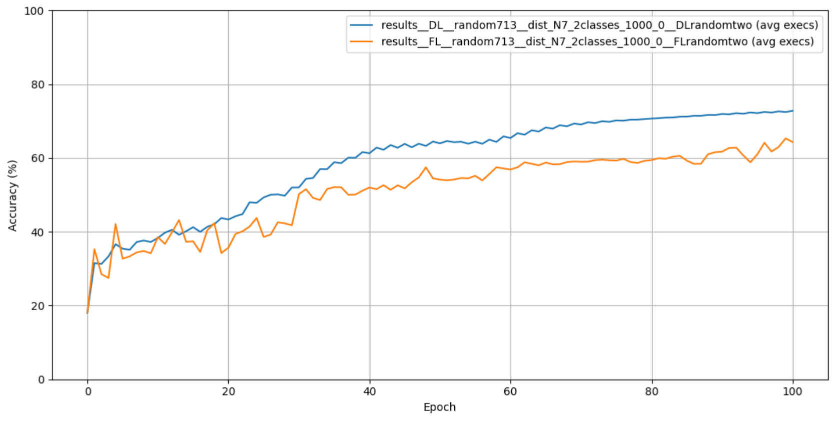

Figure 5.7: Comparison between DL and FL in a random network with Two-Class distribution.

- Random Graph, Four-Class Distribution

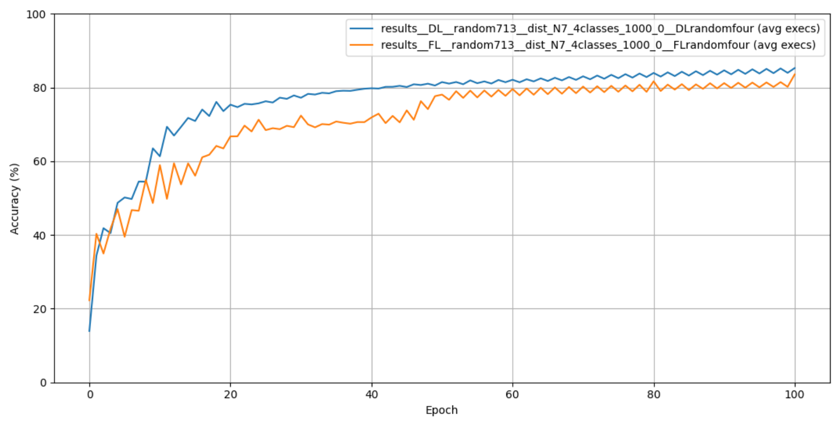

Figure 5.8: Comparison between DL and FL in a random network with Four-Class distribution.

- Random Graph, Varying-Class Distribution

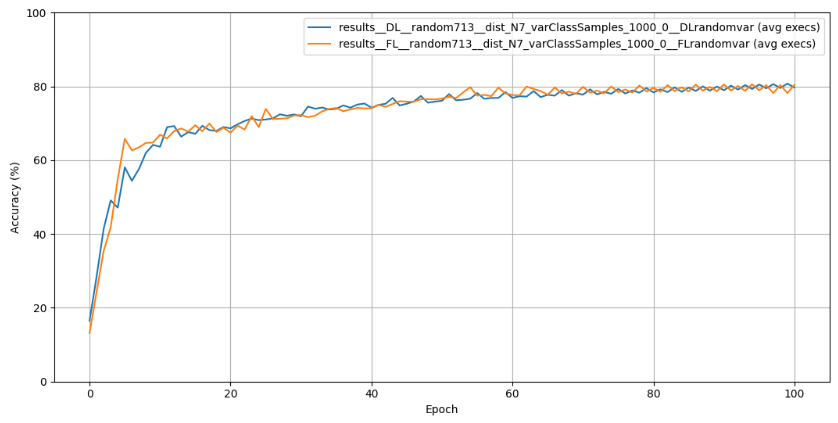

Figure 5.9: Comparison between DL and FL in a random network with Varying-Class distribution.

- Random Graph, Random Distribution

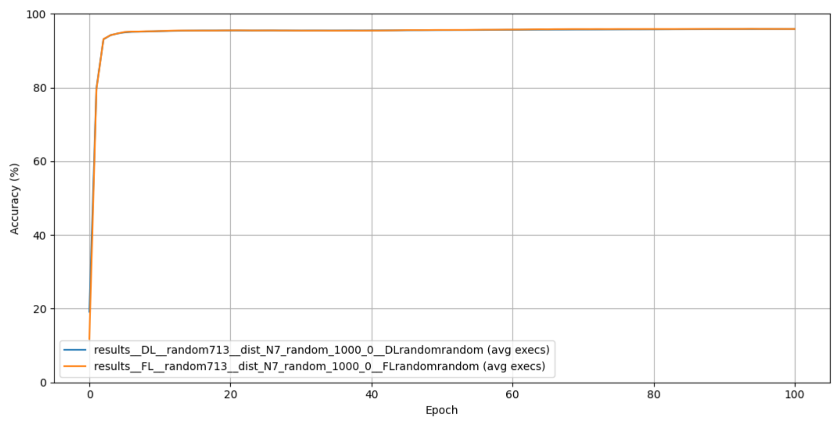

Figure 5.10: Comparison between DL and FL in a random network with random distribution.

- Mesh Graph, Two-Class Distribution

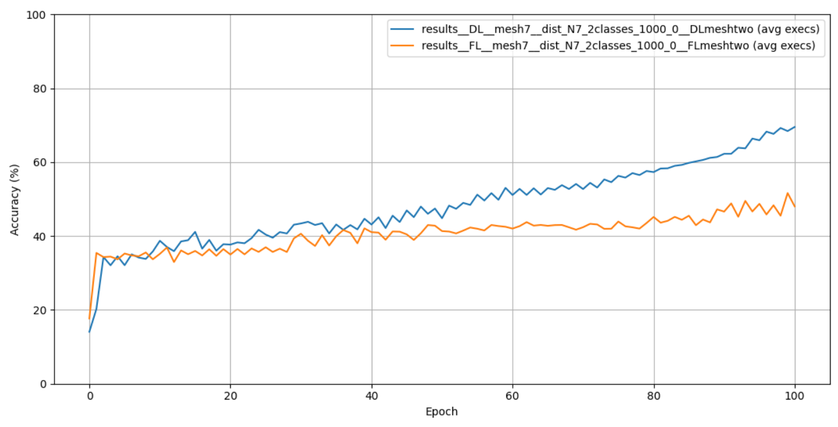

Figure 5.11: Comparison between DL and FL in a mesh network with Two-Class distribution.

- Mesh Graph, Four-Class Distribution

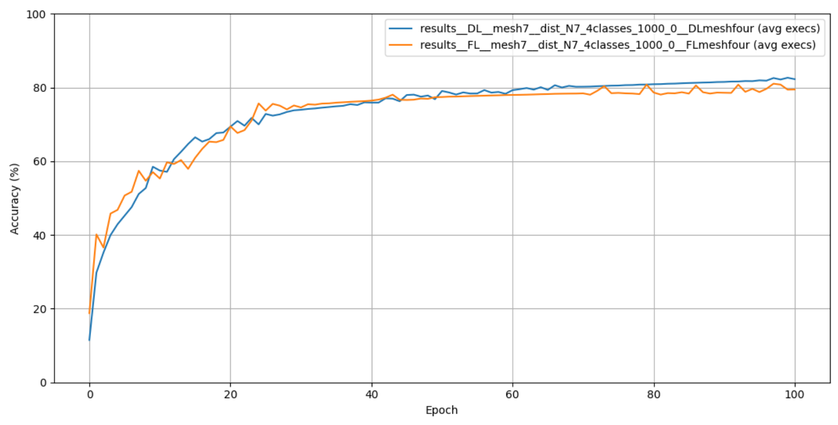

Figure 5.12: Comparison between DL and FL in a mesh network with Four-Class distribution.

- Mesh Graph, Varying-Class Distribution

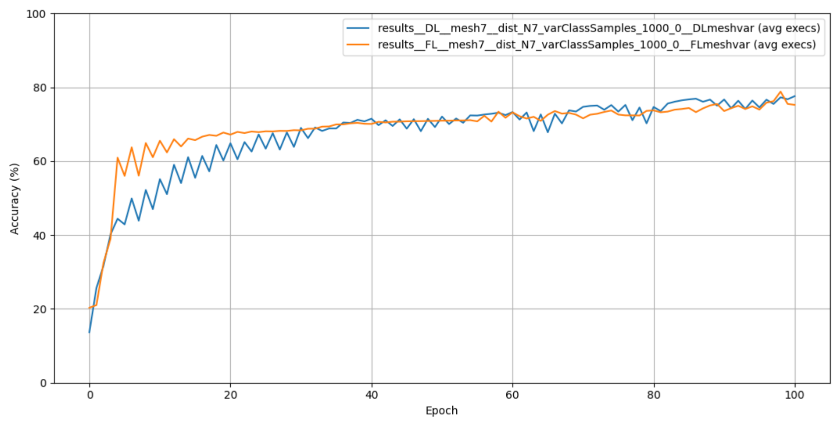

Figure 5.13: Comparison between DL and FL in a mesh network with Varying-Class distribution.

- Mesh Graph, Random Distribution

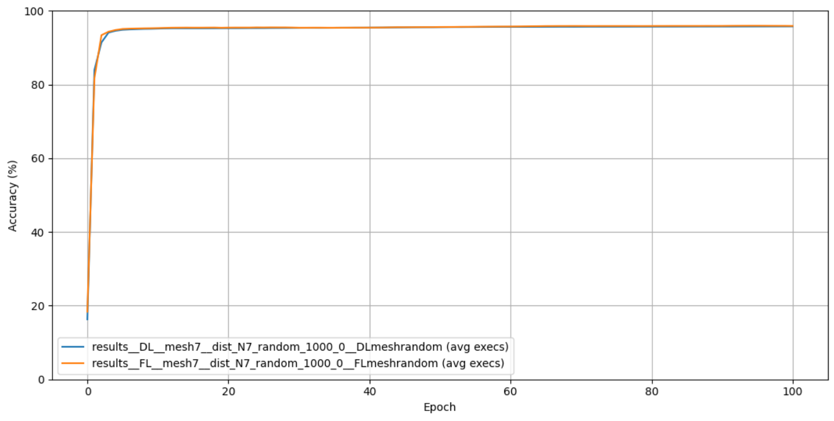

Figure 5.14: Comparison between DL and FL in a mesh network with random distribution.

- All DL Scenarios

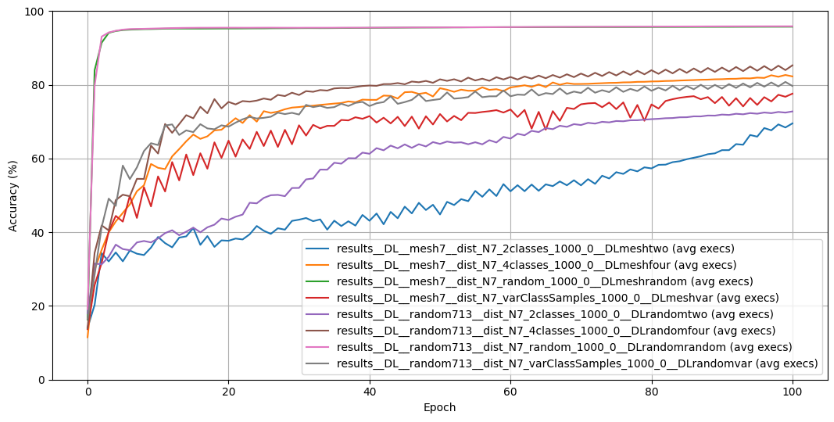

Figure 5.15: Comparison between different DL scenarios.

- All FL Scenarios

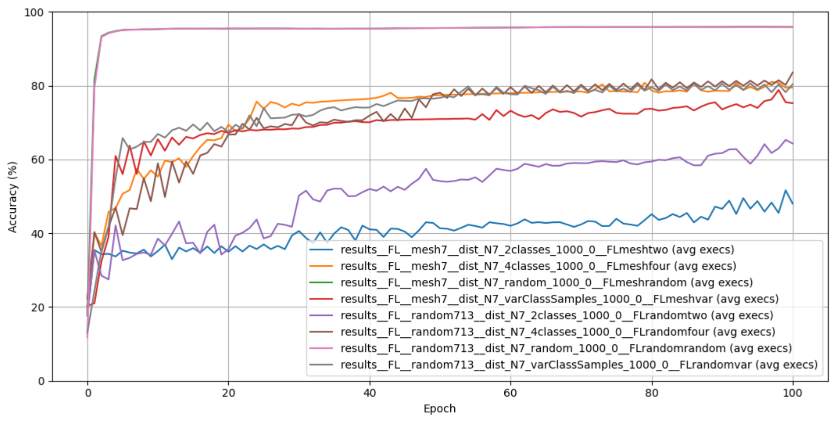

Figure 5.16: Comparison between different FL scenarios.

We can observe that as the data becomes non-IID, such as in a Two-Class distribution, DL outperforms FL. This phenomenon happens because as k decreases in the k-class distribution, the mismatch between each node's distribution and the global distribution increases. This difference causes problems in FL, because in FL the nodes train their model locally, and then the central server node aggregates all the local models and sends the global model back to the nodes. However, in a k-class distribution scenario, this can be a drawback of FL.

Let's consider a k = 2 distribution, in which node 1 has data only from class A and class B, and node 2 has data only from class C and class D. In this case, the global model returned to the nodes is suboptimal for them. This observation does not occur in DL, because in DL the exchange of model information occurs locally and is less disruptive than global averaging.

As k decreases, each node observes a smaller subset of the whole dataset, which can lead to a greater difference between the local training data on which a specific node trains its model and the entire dataset that forms the basis for the global model returned by the central server node. Consequently, that model may perform worse on the local data, either by exhibiting poorer performance metrics or slower convergence. In contrast, DL benefits from the network structure, as local nodes communicate only with their neighbors, who are more likely to have common data; or at least, if their data is heterogeneous, the information exchanged is less disruptive.

As mentioned earlier, this set of experiments was conducted on the HAR dataset, which has 6 classes, with 7 nodes in the simulated network. In a real-world scenario, the number of nodes is much higher, and the number of classes in the dataset will increase significantly, which will lead to a greater difference between the performance metrics of DL and FL. Therefore, in real-world applications, DL can be chosen as the optimal learning scheme for the network.

For both FL and DL scenarios, it can be seen that as the data loses its IID property, such as in a Two-Class or Varying-Class distribution, the average accuracy of the nodes decreases after a certain number of epochs. In other words, convergence for non-IID scenarios is slower; therefore, training for non-IID cases should run for a higher number of epochs to reach the same performance.

Also, it is worth noting that both DL and FL had nearly identical and perfect performance under IID data, which confirms that under traditional ML assumptions of IID data, learning schemes work perfectly. However, in real-world cases where heterogeneity results in the loss of the IID property, traditional assumptions and solutions are not applicable.

5.2.2 Network Graph Effect

In this section, the results for the alternating network graph are shown. As shown in Chapter 4, for this set of experiments we used a random graph and a mesh graph.

- Random Graph

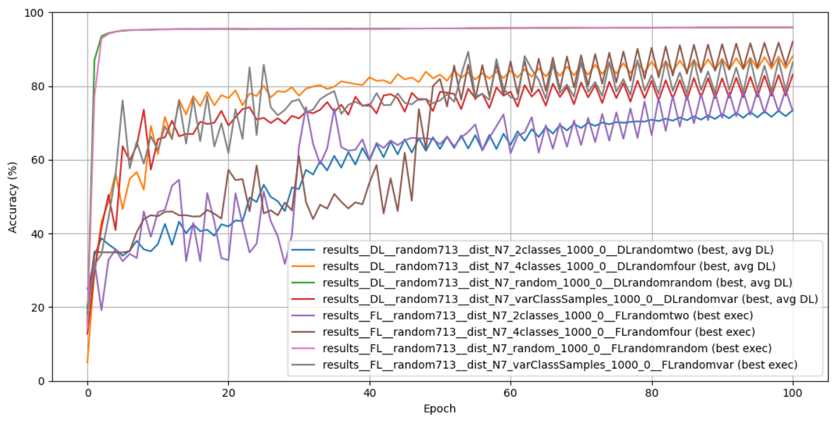

Figure 5.17: Average accuracy of the nodes in different executions in a random network graph with 7 nodes and 13 edges.

- Mesh Graph

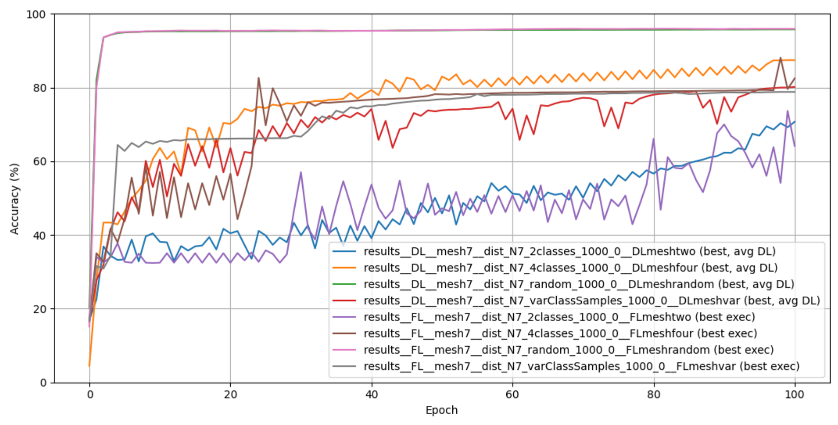

Figure 5.18: Average accuracy of the nodes in different executions in a mesh network graph with 7 nodes.

We can see that while in a random graph DL and FL have performed similarly, in a mesh graph DL has outperformed FL in all scenarios, which confirms the superiority of DL over FL in real-world scenarios. Also, it is worth noting that convergence in DL scenarios is faster, and accuracy fluctuations during training are lower than in FL scenarios.

5.2.3 Data Distribution Effect

In this part, the plots for comparing the performance of the learning schemes under different data distributions are shown.

- Random Distribution

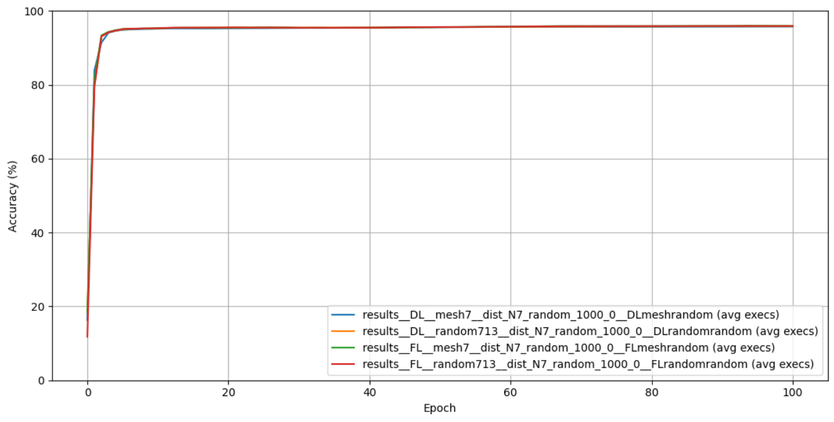

Figure 5.19: Average accuracy of the nodes in different executions in a random distribution scenario.

- Four-Class Distribution

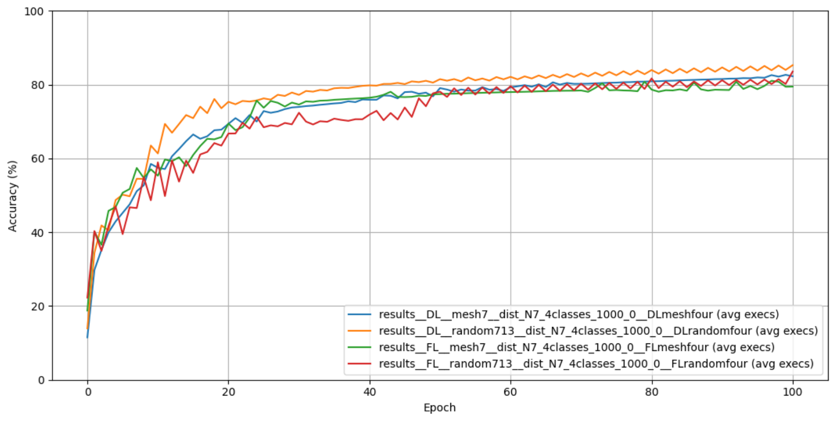

Figure 5.20: Average accuracy of the nodes in different executions in a Four-Class distribution scenario.

- Varying-Class Distribution

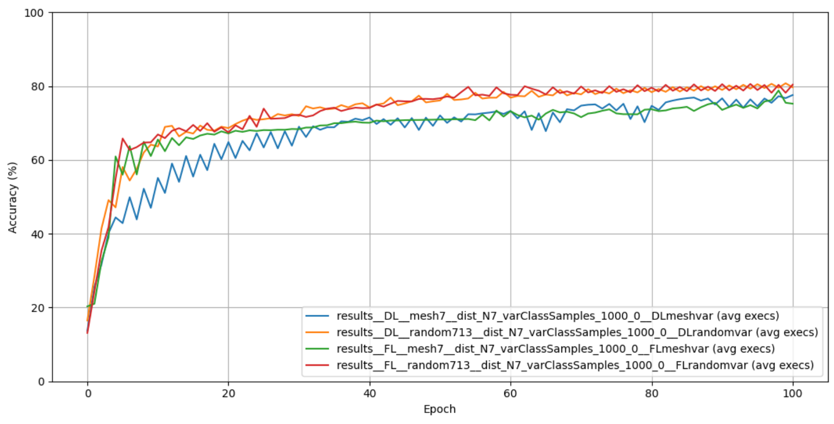

Figure 5.21: Average accuracy of the nodes in different executions in a Varying-Class distribution scenario.

- Two-Class Distribution

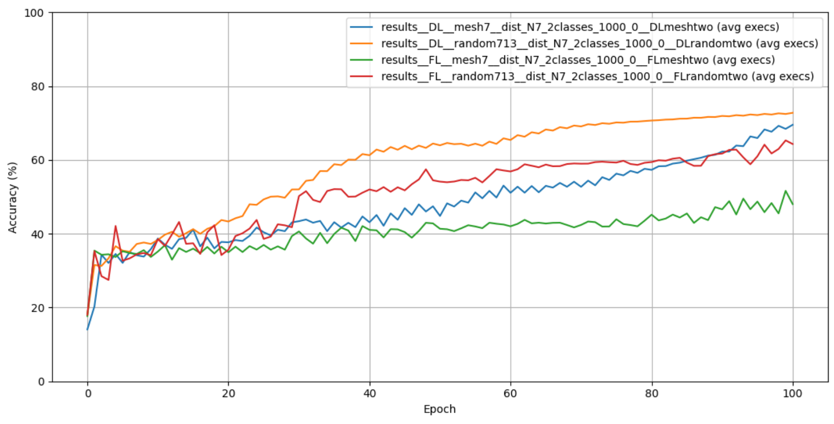

Figure 5.22: Average accuracy of the nodes in different executions in a Two-Class distribution scenario.

As we can see, as heterogeneity in the data increases, the difference between DL and FL becomes more pronounced. In the heterogeneous environment, DL accuracies are higher, and even in the Two-Class scenario, while FL performance does not improve over multiple epochs, DL performance continues to improve with each epoch, showing an increasing trend.

Chapter 6

Conclusions and Future Works

6.1 Summary of Contributions

In this thesis, our goal was to study the effect of different key factors and parameters on machine learning in 5G networks. We designed and developed various tools for graph generation, distribution generation, learning schemes, and results analysis. Different metrics for measuring heterogeneity, different methods to generate distributions, and different graphs for 5G networks were deployed and tested, and an overview of the most important achievements was shown in the previous chapter.

Based on our findings, DL has outperformed FL under different conditions. This finding stems from the fact that in FL, all nodes train their models locally with their own data and send their model to the central server node. The central server node aggregates all the received models and sends back the global model to the nodes. However, as data heterogeneity increases in real-world scenarios, this structure becomes a drawback for FL because, as data may vary significantly between nodes, a global model may not be optimal for all nodes, leading to a decrease in the performance of the final model.

However, DL is more robust against heterogeneity because in DL, each node only exchanges the weights of its local model with its neighbors. These neighbors are either similar to the original node, which leads to low heterogeneity and low disruption, or, in the worst case, if they are very diverse, the number of models that negatively affect performance is much lower than the total number of nodes; therefore, the disruption is not as severe as in FL.

6.2 Limitations

In this study, we opted to work with the HAR dataset, which was good for educational and basic purposes; however, this dataset is relatively small. To obtain better and more general results, it is preferable to switch to larger datasets to better reflect the diversity and heterogeneity of real-world scenarios. It is worth noting that the HAR dataset was suitable for proof-of-concept studies.

Another challenge we faced during these experiments was simulation constraints, such as the number of nodes or edges in the network graph. For more realistic simulations, it is better to work with more populated graphs, as in real-world applications the number of nodes in the network is higher than the number used in our scenarios.

Another constraint we faced was hardware limitations. Due to the high number of experiments we had to run because of the large number of parameters and metrics, this hardware constraint was an obstacle. For example, for rDL, more experiments need to be run to better understand the edge removal criteria. To obtain better and more generalized results, it is crucial to upgrade to more powerful hardware so that more scenarios can be tested to better mimic real-world applications.

6.3 Future Research Directions

A further step worth taking in the future is to extend and apply our tools to a real-world 5G application. For example, it can be beneficial to collect a large dataset from IoT devices and simulate real-world network conditions such as latencies, failures, and network topology. By doing so, we can learn about the shortcomings in our tools, which can lead to a better system in terms of scalability and applicability to real-world ML use cases in 5G.

For a more advanced and comprehensive study of the effect of heterogeneity, it is worth including other metrics, such as distribution skewness or kurtosis, or some domain-specific metrics. Also, some metrics can be combined or analyzed together to obtain a better understanding of performance.

Also, a possible enhancement is to add and consider some performance metrics for the overall system. There are parameters such as communication overhead and overall system latency that can be used to evaluate the performance of the overall system under different hyperparameters, especially the hyperparameters that we kept constant during our experiments, such as network topology.

Although we can see that DL outperforms FL in different scenarios, it is not a flawless approach. In order to reduce latency, network flow, and energy consumption, we propose a new method for future work, named reduced decentralized learning (rDL), which aims to maintain high performance while removing some network edges based on different criteria.

Another step to further generalize the results and improve scalability is to explore advanced learning schemes based on currently deployed methods. For example, it would be beneficial to test current schemes with gradient compression or meta-learning to evaluate their performance. Also, in the rDL scheme, it is worth testing different edge-removal strategies, or even hybrid strategies, to evaluate performance.

Appendix A

Code Snippets

The core section of data distribution generation is brought below.

```
    if dist_type == 'random':
    rng = list(range(len(y_train)))
    random.shuffle(rng)
    rng = [rng[i:i+target] for i in range(0, N*target, target)]
elif isinstance(dist_type, int):
    classes = get_classes(y_train)
    rng = []
    for r in range(N):
        r = []
        for c in random.sample(list(range(n_classes)),dist_type):
            r.extend(classes[c])
        rng.append(random.sample(r,target))
    dist = [calculate_distribution(y_train[r],n_classes) for r in rng
    ]
    _,bins=np.histogram(range(n_classes),bins=n_classes)
    plt.figure()
    plt.hist([bins[:-1] for _ in dist],weights=dist,bins=n_classes,
    histtype='barstacked')
    plt.legend([str(r) for r in range(len(dist))])
    plt.show()
elif dist_type == 'allClasses':
    classes = get_classes(y_train)
    classesIndex = set(range(n_classes))
    random.shuffle(classes)
    if N > n_classes:
        c = [random.randint(1,n_classes) for _ in range(N)]
        if sum(c) < n_classes: raise SystemError('Sum of classes is
    not equal to n_classes')
        targetClasses = classesIndex.copy()
        while targetClasses:
            targetClasses = classesIndex.copy()
            prerng = []
            for i in range(N):
                prerng.append(random.sample(list(classesIndex),c[i]))
                targetClasses -= set(prerng[-1])
        for i in range(N):
            prerng[i] = [sample for k in prerng[i] for sample in classes[k]]
    else:
        c = [random.randint(1,n_classes-(N-1))]
        for n in range(1,N-1): c.append(random.randint(1,n_classes-sum(c)-(N-1-n)))
        c.append(n_classes-sum(c))
        if sum(c) != n_classes: raise SystemError('Sum of classes is not equal to n_classes')
        prerng = []
        for i in range(N):
            targetClasses = set(classesIndex)
            for p in prerng: targetClasses -= p
            if i == N-1: prerng.append(targetClasses)
            else: prerng.append(set(random.sample(list(targetClasses),c[i])))
        for i in range(N):
            prerng[i] = list(prerng[i]) + random.sample(list(classesIndex-prerng[i]), int(random.randint(0,6-c[i])))
            prerng[i] = [sample for k in prerng[i] for sample in classes[k]]
    rng = [random.sample(prerng[i],target) for i in range(N)]
elif dist_type == 'varClassSamples':
    classes = get_classes(y_train)
    classesIndex = set(range(n_classes))
    random.shuffle(classes)
    if N > n_classes:
        c = [random.randint(1,n_classes) for _ in range(N)]
        if sum(c) < n_classes: raise SystemError('Sum of classes is not equal to n_classes')
        targetClasses = classesIndex.copy()
        while targetClasses:
            targetClasses = classesIndex.copy()
            prerng = []
            for i in range(N):
                prerng.append(random.sample(list(classesIndex),c[i]))
                targetClasses -= set(prerng[-1])
        for i in range(N):
            prerng[i] = [sample for k in prerng[i] for sample in classes[k]]
    else:
        c = [random.randint(1,n_classes-(N-1))]
        for n in range(1,N-1): c.append(random.randint(1,n_classes-sum(c)-(N-1-n)))
        c.append(n_classes-sum(c))
        if sum(c) != n_classes: raise SystemError('Sum of classes is not equal to n_classes')
        prerng = []
        for i in range(N):
            targetClasses = set(classesIndex)
            for p in prerng: targetClasses -= p
            if i == N-1: prerng.append(targetClasses)
            else: prerng.append(set(random.sample(list(targetClasses),c[i])))
        for i in range(N):
            prerng[i] = list(prerng[i]) + random.sample(list(classesIndex-prerng[i]), int(random.randint(0,6-c[i])))
            prerng[i] = [sample for k in prerng[i] for sample in classes[k]]
    rng = [random.sample(prerng[i],random.randint(target-30,target+30)) for i in range(N)]
elif dist_type == 'beta':
        rng = list(range(len(y_train)))
        rng = generate_beta_distribution(n_classes, target, N)
```

Here is the graph generation section of the code.

```
    if nw_type == 'random':
    G = nx.gnm_random_graph(N,edges)
elif nw_type == 'path':
    G = nx.path_graph(N)
elif nw_type == 'circle':
    G = nx.cycle_graph(N)
elif nw_type == 'unconnected':
    G = nx.empty_graph(N)
elif nw_type == 'complete':
    G = nx.complete_graph(N)
elif nw_type == 'star':
    G = nx.star_graph
elif nw_type == 'mesh':
    m = int(math.sqrt(N))
    while N%m > 0:
        m -= 1
    G = nx.grid_2d_graph(m,N//m)
elif nw_type == '3lollipop':
    G = nx.lollipop_graph(3, N-3)
elif nw_type == '4lollipop':
    G = nx.lollipop_graph(4, N-4)
elif nw_type == '3cliqueStar':
    G = nx.complete_graph(3)
    for i in range(3,N):
        G.add_edge(0,i)
elif nw_type == '4cliqueStar':
    G = nx.complete_graph(4)
    for i in range(4,N):
        G.add_edge(0,i)
elif nw_type == '3cliqueLeaves':
    G = nx.complete_graph(3)
    j=0
    for i in range(3,N):
        G.add_edge(j%3,i)
        j += 1
elif nw_type == '4cliqueLeaves':
    G = nx.complete_graph(4)
    j=0
    for i in range(4,N):
        G.add_edge(j%4,i)
        j += 1
```

In the following snippet, the codes for visualization are written.

```
    def plot_AccLoss_loop_avgExecutions_by_node(
    fileNames,
    isAcc=True,
    vsTime=True,
    x_label=None,
    y_label=None
):
    global ACC_INDEX, LOSS_INDEX, TIME_INDEX
    metric_index = ACC_INDEX if isAcc else LOSS_INDEX

    if x_label is None:
        x_label = "Time" if vsTime else "Epoch"
    if y_label is None:
        y_label = "Accuracy (%)" if isAcc else "Loss"

    for n in fileNames:
        subString = n.split('__')
        mode = subString[1] if len(subString) >= 2 else "Unknown"

        with open(path + n, 'rb') as f:
            res = pickle.load(f)

        if 'DL' in mode or 'rDL' in mode:
            num_executions = len(res)
            num_nodes = len(res[0])

            plt.figure()
            for j in range(num_nodes):
                node_curves = []
                for i in range(num_executions):
                    node_curve = res[i][j][metric_index]
                    node_curves.append(node_curve)
                node_curves = np.array(node_curves)
                avg_curve = node_curves.mean(axis=0)
                time_axis = res[0][j][TIME_INDEX]
                if vsTime:
                    plt.plot(time_axis, avg_curve, label=f'Node {j}')
                else:
                    plt.plot(avg_curve, label=f'Node {j}')
            plt.title(n)
            plt.xlabel(x_label)
            plt.ylabel(y_label)
            plt.grid()
            plt.legend()
            if isAcc:
                plt.ylim(0, 100)
            plt.show()
            plt.close()
        else:
            plt.figure()
            curves = []
            for exec_data in res:
                curves.append(exec_data[metric_index])
            curves = np.array(curves)
            avg_metric = curves.mean(axis=0)
            time_axis = res[0][TIME_INDEX]
            if vsTime:
                plt.plot(time_axis, avg_metric, label=n)
            else:
                plt.plot(avg_metric, label=n)
            plt.title(n)
            plt.xlabel(x_label)
            plt.ylabel(y_label)
            plt.grid()
            plt.legend()
            if isAcc:
                plt.ylim(0, 100)
            plt.show()
            plt.close()
```

Bibliography

[1] Emna Hajlaoui, Aida Zaier, Abdelhakim Khlifi, Jihed Ghodhbane, Mouna Ben Hamed, and Lassâad Sbita. «4G and 5G technologies: A Comparative Study». In: *2020 5th International Conference on Advanced Technologies for Signal and Image Processing (ATSIP)*. 2020, pp. 1–6. DOI: 10.1109/ATSIP49331.2020.9231605 (cit. on p. 1).

[2] Mohsen Attaran. «The impact of 5G on the evolution of intelligent automation and industry digitization». In: *Journal of Ambient Intelligence and Humanized Computing* 14.5 (May 2023), pp. 5977–5993. ISSN: 1868-5145. DOI: 10.1007/s12652-020-02521-x. URL: https://doi.org/10.1007/s12652-020-02521-x (cit. on p. 1).

[3] Afra Domeke, Bruno Cimoli, and Idelfonso Tafur Monroy. «Integration of Network Slicing and Machine Learning into Edge Networks for Low-Latency Services in 5G and beyond Systems». In: *Applied Sciences* 12.13 (2022). ISSN: 2076-3417. DOI: 10.3390/app12136617. URL: https://www.mdpi.com/2076-3417/12/13/6617 (cit. on p. 1).

[4] Shanchen Pang, Nuanlai Wang, Min Wang, Sibo Qiao, Xue Zhai, and Neal N. Xiong. «A Smart Network Resource Management System for High Mobility Edge Computing in 5G Internet of Vehicles». In: *IEEE Transactions on Network Science and Engineering* 8.4 (2021), pp. 3179–3191. DOI: 10.1109/TNSE.2021.3106955 (cit. on p. 1).

[5] Olaonipekun Oluwafemi Erunkulu, Adamu Murtala Zungeru, Caspar K. Lebekwe, Modisa Mosalaosi, and Joseph M. Chuma. «5G Mobile Communication Applications: A Survey and Comparison of Use Cases». In: *IEEE Access* 9 (2021), pp. 97251–97295. DOI: 10.1109/ACCESS.2021.3093213 (cit. on p. 1).

[6] Maryam Ben Driss, Essaid Sabir, Halima Elbiaze, and Walid Saad. *Federated Learning for 6G: Paradigms, Taxonomy, Recent Advances and Insights*. 2023. arXiv: 2312.04688 [cs.LG]. URL: https://arxiv.org/abs/2312.04688 (cit. on p. 3).

[7] Mohammad Abrar Shakil Sejan, Md Habibur Rahman, Md Abdul Aziz, Rana Tabassum, Young-Hwan You, Duck-Dong Hwang, and Hyoung-Kyu Song. «Interference Management for a Wireless Communication Network Using a Recurrent Neural Network Approach». In: *Mathematics* 12.11 (2024). ISSN: 2227-7390. DOI: `10.3390/math12111755`. URL: `https://www.mdpi.com/2227-7390/12/11/1755` (cit. on p. 7).

[8] Chunlu Chen, Ji Liu, Haowen Tan, Xingjian Li, Kevin Wang, Peng Li, Kouichi Sakurai, and Dejing Dou. *Trustworthy Federated Learning: Privacy, Security, and Beyond.* Nov. 2024. DOI: `10.48550/arXiv.2411.01583` (cit. on p. 7).

[9] Rehan Khan, Umer Saeed, and Insoo Koo. «FedLSTM: A Federated Learning Framework for Sensor Fault Detection in Wireless Sensor Networks». In: *Electronics* 13.24 (2024). ISSN: 2079-9292. DOI: `10.3390/electronics13244907`. URL: `https://www.mdpi.com/2079-9292/13/24/4907` (cit. on p. 7).

[10] Aymen Rayane Khouas, Mohamed Reda Bouadjenek, Hakim Hacid, and Sunil Aryal. «Training Machine Learning models at the Edge: A Survey». In: *ArXiv* abs/2403.02619 (2024). URL: `https://api.semanticscholar.org/CorpusID:268247434` (cit. on p. 7).

[11] Qimei Cui et al. *Overview of AI and Communication for 6G Network: Fundamentals, Challenges, and Future Research Opportunities.* 2025. arXiv: `2412.14538 [cs.NI]`. URL: `https://arxiv.org/abs/2412.14538` (cit. on p. 8).

[12] Ashish Rauniyar, Desta Haileselassie Hagos, Debesh Jha, Jan Håkegård, Danda B Rawat, and Vladimir Vlassov. «Federated Learning for Medical Applications: A Taxonomy, Current Trends, Challenges, and Future Research Directions». In: *IEEE Internet of Things Journal* PP (Nov. 2023), pp. 1–1. DOI: `10.1109/JIOT.2023.3329061` (cit. on p. 8).

[13] Betul Yurdem, Murat Kuzlu, Mehmet Kemal Gullu, Ferhat Ozgur Catak, and Maliha Tabassum. «Federated learning: Overview, strategies, applications, tools and future directions». In: *Heliyon* 10.19 (2024), e38137. ISSN: 2405-8440. DOI: `https://doi.org/10.1016/j.heliyon.2024.e38137`. URL: `https://www.sciencedirect.com/science/article/pii/S2405844024141680` (cit. on p. 8).

[14] Minghong Wu, Minghui Liwang, Yuhan Su, Li Li, Seyyedali Hosseinalipour, Xianbin Wang, Huaiyu Dai, and Zhenzhen Jiao. *Towards Seamless Hierarchical Federated Learning under Intermittent Client Participation: A Stagewise Decision-Making Methodology.* 2025. arXiv: `2502.09303 [cs.LG]`. URL: `https://arxiv.org/abs/2502.09303` (cit. on p. 9).

[15] Chen Qiu, Ziang Wu, Haoda Wang, Qinglin Yang, Yu Wang, and Chunhua Su. «Hierarchical Aggregation for Federated Learning in Heterogeneous IoT Scenarios: Enhancing Privacy and Communication Efficiency». In: *Future Internet* 17.1 (2025). ISSN: 1999-5903. DOI: `10.3390/fi17010018`. URL: `https://www.mdpi.com/1999-5903/17/1/18` (cit. on p. 9).

[16] Paul Vanhaesebrouck, Aurélien Bellet, and Marc Tommasi. «Decentralized Collaborative Learning of Personalized Models over Networks». In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. Ed. by Aarti Singh and Jerry Zhu. Vol. 54. Proceedings of Machine Learning Research. PMLR, 20–22 Apr 2017, pp. 509–517. URL: `https://proceedings.mlr.press/v54/vanhaesebrouck17a.html` (cit. on p. 9).

[17] István Hegedűs, Gábor Danner, and Márk Jelasity. «Decentralized learning works: An empirical comparison of gossip learning and federated learning». In: *Journal of Parallel and Distributed Computing* 148 (2021), pp. 109–124. ISSN: 0743-7315. DOI: `https://doi.org/10.1016/j.jpdc.2020.10.006`. URL: `https://www.sciencedirect.com/science/article/pii/S0743731520303890` (cit. on pp. 9, 10).

[18] Qazi Waqas Khan, Anam Nawaz Khan, Atif Rizwan, Rashid Ahmad, Salabat Khan, and Do-Hyeun Kim. «Decentralized Machine Learning Training: A Survey on Synchronization, Consolidation, and Topologies». In: *IEEE Access* 11 (2023), pp. 68031–68050. DOI: `10.1109/ACCESS.2023.3284976` (cit. on p. 9).

[19] Jun Li, Yumeng Shao, Kang Wei, Ming Ding, Chuan Ma, Long Shi, Zhu Han, and H. Vincent Poor. «Blockchain Assisted Decentralized Federated Learning (BLADE-FL): Performance Analysis and Resource Allocation». In: *IEEE Transactions on Parallel and Distributed Systems* 33.10 (2022), pp. 2401–2415. DOI: `10.1109/TPDS.2021.3138848` (cit. on p. 9).

[20] Longbing Cao. «Beyond i.i.d.: Non-IID Thinking, Informatics, and Learning». In: *IEEE Intelligent Systems* 37.4 (2022), pp. 5–17. DOI: `10.1109/MIS.2022.3194618` (cit. on p. 9).

[21] Jie Zhang, Song Guo, Zhihao Qu, Deze Zeng, Yufeng Zhan, Qifeng Liu, and Rajendra Akerkar. «Adaptive Federated Learning on Non-IID Data With Resource Constraint». In: *IEEE Transactions on Computers* 71.7 (2022), pp. 1655–1667. DOI: `10.1109/TC.2021.3099723` (cit. on p. 9).

[22] Xiaodong Ma, Jia Zhu, Zhihao Lin, Shanxuan Chen, and Yangjie Qin. «A state-of-the-art survey on solving non-IID data in Federated Learning». In: *Future Generation Computer Systems* 135 (2022), pp. 244–258. ISSN: 0167-739X. DOI: `https://doi.org/10.1016/j.future.2022.05.003`. URL: `https:`

//www.sciencedirect.com/science/article/pii/S0167739X22001686 (cit. on p. 10).

[23] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. *Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent.* 2017. arXiv: 1705.09056 [math.OC]. URL: https://arxiv.org/abs/1705.09056 (cit. on p. 10).

[24] István Hegedűs, Gábor Danner, and Márk Jelasity. «Decentralized learning works: An empirical comparison of gossip learning and federated learning». In: *Journal of Parallel and Distributed Computing* 148 (Feb. 2021), pp. 109–124. DOI: 10.1016/j.jpdc.2020.10.006 (cit. on p. 10).

[25] Yan Sun, Li Shen, and Dacheng Tao. *Towards Understanding Generalization and Stability Gaps between Centralized and Decentralized Federated Learning.* 2024. arXiv: 2310.03461 [cs.LG]. URL: https://arxiv.org/abs/2310.03461 (cit. on pp. 10, 11).