TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
FIELD OF SOFTWARE ENGINEERING

Federated Learning at Flower and NS3 Integrating the Geometric Approach for Efficient Synchronization

DIPLOMA THESIS

of

PANAGIOTIS G. SKLAVOS

Supervisor: Antonios Deligiannakis
Professor

Committee: Vasilios Samoladas
Assosiate Professor

Committee: Nikolaos Giatrakos
Assistant Professor

Chania, July 2024

TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
FIELD OF SOFTWARE ENGINEERING

Federated Learning at Flower and NS3 Integrating the Geometric Approach for Efficient Synchronization

DIPLOMA THESIS

of

PANAGIOTIS G. SKLAVOS

Approved by the examination committee on 19th July 2024.

(Signature)	*(Signature)*	*(Signature)*
.....	.....	.....
A. Deligiannakis	V. Samoladas	N. Giatrakos
Professor	Assosiate Professor	Assistant Professor

Chania, July 2024

TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
FIELD OF SOFTWARE ENGINEERING

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 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.

(Signature)

.....
Panagiotis G. Sklavos

19th July 2024

Abstract

The rapid advancement of edge computing and the Internet of Things (IoT) has led to an exponential increase in data generation, underscoring the need for privacy-preserving and efficient decentralized machine learning methods. This thesis addresses these needs by implementing Federated Learning (FL) under realistic network conditions, integrating the Flower Framework with the NS3 network simulator, and employing the Geometric Method (GM) for enhanced synchronization and performance. Our approach involves several key steps. Initially, a Federated Learning orchestrator is developed using the Flower framework to establish a distributed network with a central server and multiple clients connected in a star topology. To optimize model updates and minimize communication costs, a synchronization method based on geometric monitoring, known as Functional Dynamic Averaging (FDA), is implemented. Additionally, the NS3 network simulator is used to emulate realistic network conditions, and a socket-based Inter-Process Communication (IPC) protocol is employed to ensure seamless interaction between the federated learning framework and the network simulator. Our integrated FL framework demonstrates robustness and effective synchronization across various simulated network conditions. The Geometric Monitoring approach of FDA efficiently balances the computation-to-communication ratio while maintaining high accuracy levels. Thorough testing across diverse datasets, artificial neural networks (ANNs), networking conditions, and data distributions (IID and non-IID) reveals significant improvements in communication efficiency and model accuracy compared to a baseline distributed algorithm. In conclusion, this research presents a novel and effective Federated Learning framework that bridges existing infrastructure gaps, ensuring robust performance and efficient synchronization in real-world network environments.

Keywords

Online Machine Learning (OML), Edge Computing, Artificial Neural Networks (ANNs), Distributed Systems, Distributed Machine Learning (DML), Data Privacy, Federated Learning (FL), Synchronization Techniques, Geometric Monitoring (GM), Functional Dynamic Averaging (FDA), Flower Framework, NS3 Network Simulator, Communication Efficiency,

Περίληψη

Η ταχεία εξέλιξη του υπολογιστικού περιβάλλοντος αιχμής (edge computing) και του Διαδικτύου των Πραγμάτων (IoT) έχει οδηγήσει σε εκθετική αύξηση του παραγώμενου όγκου δεδομένων, επισημαίνοντας την ανάγκη για μεθόδους αποκεντρωμένης μηχανικής μάθησης που διασφαλίζουν ιδιωτικότητα και απόδοση. Η παρούσα διπλωματική εργασία ανταποκρίνεται σε αυτές τις ανάγκες, εφαρμόζοντας Ομοσπονδιακή Μάθηση (FL) υπό ρεαλιστικές δικτυακές συνθήκες, συνδυάζοντας το πλαίσιο Flower με τον προσωμοιωτή δικτύου NS3 και χρησιμοποιώντας τη Γεωμετρική Μέθοδο για βελτιωμένο συγχρονισμό και επιδόσεις. Η προσέγγισή μας περιλαμβάνει πολλαπλά βασικά βήματα. Αρχικά, αναπτύχθηκε ένας ορχηστρωτής Ομοσπονδιακής Μάθησης χρησιμοποιώντας το πλαίσιο Flower για τη δημιουργία ενός κατανεμημένου δικτύου με έναν κεντρικό διακομιστή και πολλαπλούς πελάτες συνδεδεμένους σε τοπολογία αστέρα. Για τη βελτιστοποίηση των ενημερώσεων των νευρωνικών δικτύων και τον περιορισμό των επικοινωνιακών εξόδων, εφαρμόστηκε μια μέθοδος συγχρονισμού βασισμένη στη γεωμετρική παρακολούθηση, γνωστή ως Functional Dynamic Averaging (FDA). Επιπλέον, χρησιμοποιήθηκε ο προσομοιωτής δικτύου NS3 για την εξομείωση ρεαλιστικών συνθηκών δικτύου και εφαρμόστηκε ένα πρωτόκολλο Διαδιεργασιακής Επικοινωνίας (IPC) με χρήση Unix Sockets για την απρόσκοπτη αλληλεπίδραση μεταξύ του πλαισίου Ομοσπονδιακής Μάθησης και του προσομοιωτή δικτύου. Το ολοκληρωμένο πλαίσιο Ομοσπονδιακής Μάθησης που υλοποιήθηκε παρουσιάζει ανθεκτικότητα και αποτελεσματικό συγχρονισμό σε ποικιλόμορφες εξομοιωμένες συνθήκες δικτύου. Η Γεωμετρική προσέγγιση που εφαρμόζεται από τον αλγόριθμο FDA εξισορροπεί αποτελεσματικά τον λόγο υπολογισμού προς το επικοινωνιακό κόστος, διατηρώντας υψηλά επίπεδα ακρίβειας. Εξονυχιστικές δοκιμές σε ποικίλα σύνολα δεδομένων, τεχνητά νευρωνικά δίκτυα (ANN), συνθήκες δικτύου και κατανομές δεδομένων (δεδομένα ανεξάρτητα και ομοιόμορφα κατανεμημένα και μη) αποκαλύπτουν σημαντικές βελτιώσεις στην αποδοτικότητα επικοινωνίας και την ακρίβεια του μοντέλου σε σύγκριση με έναν βασικό κατανεμημένο αλγόριθμο. Συμπερασματικά, αυτή η έρευνα παρουσιάζει ένα καινοτόμο και αποδοτικό πλαίσιο Ομοσπονδιακής Μάθησης που γεφυρώνει τα υπάρχοντα κενά σε υποδομές ανάπτυξης και προσωμοίωσης, εξασφαλίζοντας ανθεκτική απόδοση και αποτελεσματικό συγχρονισμό σε πραγματικές ρεαλιστικές συνθήκες δικτύου.

Λέξεις Κλειδιά

Μηχανική Μάθηση σε Πραγματικό Χρόνο, Υπολογιστικό Περιβάλλον Αιχμής, Τεχνητά Νευρωνικά Δίκτυα, Κατανεμημένα Συστήματα, Μάθηση σε Κατενεμημένα Συστήματα, Προστασία Προσωπικών δεδομένων, Ομοσπονδιακή Μάθηση, Μέθοδοι Συγχρονισμένης Μάθησης, Γεωμετρική Μέθοδος, Functional Dynamic Averaging (FDA), πλαίσιο Flower, NS3 Προσωμοιωτής Δικτύου, Αποδοτική Επικοινωνία,

Acknowledgements

First and foremost, I am deeply grateful to my supervising professor, Mr. Antonios Deligiannakis, for providing me with the opportunity to work on this captivating research project. His unwavering support, timely guidance, and extensive knowledge have been instrumental in the completion of this thesis. His innovative research approach and passion for the field have inspired me to delve deeper into this area, igniting a genuine interest and enthusiasm for the subject matter.

I would also like to extend my heartfelt thanks to the members of the committee: Professor Vasilis Samoladas, for his insightful suggestions, sincere input, and steadfast interest in my work. Additionally, I am thankful to Professor Nikos Giatrakos for his participation in the committee, and his keen research acumen and thoughtful consideration of my work.

I am immensely appreciative of the support provided by the research team at SoftNet Lab of the Technical University of Crete. The collaborative efforts with faculty members and fellow students have played a crucial role in advancing this research. I would like to especially acknowledge Giorgos Frangias, whose assistance in numerous instances significantly accelerated my progress and helped me navigate various challenges.

On a personal note, I am profoundly thankful to my family—my parents and brother. Their unwavering love, support, and encouragement have been a constant source of strength and motivation. I am sincerely grateful for the countless opportunities they have provided me, and for always standing by my side. Words cannot fully express the depth of my gratitude and sincere admiration towards them, as they are special in every way imaginable.

Finally, I want to thank my friends for standing by me through thick and thin, providing companionship and support, and creating countless cherished memories. The strong relationships and unforgettable experiences we've shared will always be a part of me. A special mention to Ariadne, whose presence and unwavering support have been a constant source of comfort and inspiration throughout this beautiful adventure.

This journey would not have been possible without each of you. Thank you all for your contributions and support.

Chania, July 2024

Panagiotis G. Sklavos

Contents

Abstract vi

Περίληψη vii

Acknowledgements viii

List of Abbreviations xv

1 Introduction 1

1.1 Gaps And Motivation . . . 2
1.2 Scientific Contributions and Employed Approach . . . 3
1.3 Thesis Outline . . . 4

I Literature Review 5

2 Theoretical Background 6

2.1 Fundamentals of Machine Learning . . . 6
2.1.1 Bridging Machine Learning and Artificial Intelligence . . . 6
2.1.2 The Taxonomy Machine Learning Techniques . . . 6
2.1.3 Model Complexity & Challenges . . . 9
2.2 Artificial Neural Networks . . . 10
2.2.1 Perceptron and Activation Functions . . . 10
2.2.2 Network Architectures . . . 13
2.2.3 Learning in Neural Networks . . . 16
2.3 Optimization Techniques . . . 17
2.3.1 Gradient Descent And Its Variants . . . 17
2.3.2 Towards ADAM Optimizer . . . 19
2.4 Distributed Machine Learning . . . 21
2.4.1 The Need for Distribution . . . 21
2.4.2 Distributed Machine Learning Architecture . . . 22
2.4.3 Challenges in Distributed Machine Learning . . . 25
2.5 Federated Learning . . . 26
2.5.1 Principles of Federated Learning . . . 27
2.5.2 Federated Learning Categorizations . . . 28
2.5.3 Federated Learning Methodology . . . 29
2.5.4 Problem Formulation in Federated Learning . . . 30
2.6 Networking in Federated Systems . . . 33
2.6.1 Network Topologies . . . 33
2.6.2 Communication in Federated Systems . . . 33
2.6.3 Communication Performance Metrics . . . 35
2.6.4 Simulating Real-World Networks . . . 36

2.7 Summary & Transition to Related Work . . . 37

3 Related Work **38**
3.1 Datasets in Federated Learning Research . . . 38
3.1.1 Benchmark Datasets . . . 38
3.1.2 Real-world Datasets . . . 39
3.1.3 IID VS. non-IID Data Distributions . . . 40
3.2 CNN Architectures . . . 41
3.2.1 LeNet-5 . . . 41
3.2.2 VGG16 . . . 41
3.2.3 ResNet . . . 42
3.3 Federated Learning Algorithms . . . 43
3.3.1 Federated Stochastic Gradient Descent (FedSGD) . . . 43
3.3.2 Federated Averaging (FedAvg) . . . 44
3.3.3 Contempory Challenges of Federated Learning . . . 45
3.3.4 Countermeasures Introduced in Thesis . . . 46
3.4 Synchronization Techniques . . . 47
3.4.1 Geometric Method . . . 47
3.4.2 Functional Dynamic Averaging . . . 51
3.4.3 Linear Approximation . . . 55
3.4.4 Sketch FDA . . . 56
3.4.5 Multiprogramming &Synchronization Primitives in DML . . . 56
3.5 NS3-FL: Bridging Data & Network Management . . . 58
3.5.1 Framework Overview . . . 58
3.5.2 Learning, Network, and Power Models . . . 58
3.5.3 Implementation . . . 59
3.5.4 Simulation Results . . . 59
3.5.5 Conclusion . . . 60

II Technical Implementation **61**

4 Environment Replication **62**
4.1 Tensorflow & Keras Deep Learning Platforms . . . 62
4.1.1 TensorFlow Framework . . . 62
4.1.2 Keras Library . . . 63
4.2 Flower Framework: An Efficient Approach to FL . . . 63
4.3 NS3: An Event-Driven Network Simulator . . . 65
4.4 Posix Sockets & gRPC . . . 66
4.4.1 Posix Sockets . . . 66
4.4.2 gRPC . . . 66

5 Methodology **67**
5.1 System Architecture . . . 67
5.2 Dataset Preprocessing & Management . . . 69
5.3 Model Development and Training . . . 70
5.3.1 Default Model Architectures . . . 71
5.3.2 Custom Training Logic for FDA . . . 72
5.4 Federated Learning Implementation . . . 72
5.4.1 Synchronization Approaches . . . 72

5.4.2 Employement of gRPC Communication Protocol . . . 74
5.5 Network Simulator Integration . . . 75
5.5.1 NS3 Program Component Analysis . . . 75
5.5.2 Holistic Workflow Post-Integration . . . 76
5.6 Evaluation Metrics and Performance Analysis . . . 77
5.6.1 Model Evaluation Metrics . . . 77
5.6.2 Network Simulation Metrics . . . 78
5.6.3 Performance Analysis . . . 78
5.6.4 Metrics Processing . . . 78

6 Experimental Results &Discussion **79**
6.1 Overview of Experiments . . . 79
6.2 Experimental Infrastracture . . . 80
6.3 Experiments Configuration . . . 81
6.4 Findings and Analysis . . . 83
6.4.1 Comparison of FDA vs. Online SyncSGD Approach . . . 83
6.4.2 Impact of Varying Thresholds . . . 86
6.4.3 Diversity of Datasets and ANN Architectures . . . 89
6.4.4 Handling Non-IID Data Distributions . . . 90
6.4.5 Diverse Network Condition Scenarios . . . 94
6.4.6 Communication Decomposition . . . 96

III Epilogue **98**

7 Conclusions and Future Work **99**
7.1 Research Conclusions . . . 99
7.2 Future Work . . . 100
7.2.1 Real World Verification of the Results . . . 100
7.2.2 Extending Heterogeneous Testing of FDA . . . 100
7.2.3 Additional Testing on More Advanced ANNs . . . 100
7.2.4 Optimization Regarding NS3 Simulations . . . 101
7.2.5 Computation Capacity Simulation . . . 101

Appendices **102**

A Default Model Architectures **103**

Bibliography **109**

List of Figures

2.1 Multi-dimensional taxonomy of Machine Learning . . . 7
2.2 Rosenblatt's perceptron architecture . . . 11
2.3 Multi-Layer Perceptron (MLP) architecture with n input neurons, one hidden layer with $h_1 = 3$ neurons, and $m = 1$ output neuron . . . 12
2.4 Side by side illustration of FFNN and RNN architectures . . . 14
2.5 CNN processing for handwritten digits classification . . . 14
2.6 Convolution of a 4×4 image with a 3×3 kernel highlighting the sliding window operation . . . 15
2.7 Demosntration of Max and Average pooling operations . . . 15
2.8 Gradient Descent Algorithm . . . 17
2.9 Impact of unsuccessful learning rate selection . . . 19
2.10 Trajectories in ravine surfaces, with and without momentum . . . 19
2.11 Comparison between centralized and distributed learning approaches . . . 21
2.12 Data and Model Parallelism methods . . . 22
2.13 Principle Distributed System communication topologies . . . 23
2.14 Federated Learning Characteristics . . . 27
2.15 Federated Learning classification based on the learning scale . . . 28
2.16 Federated Learning classification based on alignment of distributed data . . . 29
2.17 General Federated Learning training process . . . 31

3.1 Features of a good dataset for Machine Learning . . . 38
3.2 Illustration of Fashion-MNIST samples . . . 39
3.3 Example of IID and non-IID distributions of MNIST . . . 40
3.4 LeNet-5 architecture . . . 41
3.5 VGG16 architecture . . . 42
3.6 ResNet50 architecture with residual learning block illustration . . . 43
3.7 Drift vectors comprising the convex hull . . . 50
3.8 Architectural overview of NS3-FL . . . 58
3.9 Synchronous Federated Learning training workflow . . . 60

4.1 Flower architecture with one server, two edge, and one virtual clients . . . 64

5.1 Top-level architecture combining Flower Framework for Federated Learning with NS3 for network simulation . . . 68
5.2 FDA Round meeting the RTC threshold . . . 73
5.3 FDA Round breaching the RTC threshold . . . 73
5.4 NS3's architecture decomposition . . . 75
5.5 Implemented workflow depicted in a holistic sequence diagram . . . 76

6.1 Client topologies for different client populations generated with NetAnim . . . 82
6.2 Accuracy over Time plots comparing FDA with OSyncSGD on diverse client populations 83

6.3 Stacked bar diagrams comparing the Computation and Communication time contributions for FDA and OSyncSGD over varying client populations 85
6.4 Pie diagrams comparing the communication time decomposition for FDA and OSyncSGD with 20 clients 85
6.5 Combined plots of Accuracy over Time and Rounds for various RTC threshold assignments 87
6.6 Steps per communication round 87
6.7 Stacked bar diagram representation of Computation and Communication times . . 88
6.8 FDA tested on diverse ANN-dataset combinations for MNIST and CIFAR-10 89
6.9 Class percentage examples of the non-i.i.d templates distributions 90
6.10 Plots containing combined: local validation accuracy, distributed accuracy and centralized accuracy for varying heterogeneity levels and client populations 91
6.11 Plots containing combined the centralized accuracies for 5 and 15 clients for varying heterogeneity levels 93
6.12 Accuracy over Training Time plots, comparing stationary and mobile clients under varying network conditions 94
6.13 Computation to Communication Time comparison in stacked bar diagrams comparing stationary and mobile clients under varying network conditions 95
6.14 Communication Time decomposition in stacked bar diagrams comparing stationary and mobile clients under varying network conditions 97

A.1 MNIST Default ANN 103
A.2 FashionMNIST Default ANN 104

List of Tables

2.1 Learning types based on feedback along with key examples of each field 8
2.2 Commonly employed activation functions in neural networks 13
2.3 Challenges of Distributed Machine Learning 26

3.1 Notation used in Federated Learning algorithms 43
3.2 Summary of socket commands used in NS3-FL 59
3.3 Comparison of computational time (in seconds) between real deployment and NS3-FL simulations on diverse configurations 60

5.1 Supported Datasets and ANNs 71

6.1 Hardware characteristics of the computational environment 80
6.2 Federated Learning orchestrator parameter configurations 81
6.3 Naive FDA experiment configurations 81
6.4 Network simulation parameters for various WiFi configurations 82
6.5 Computation and Communication related stats 85
6.6 Communication Metrics computed for diverse network template and client mobility setups 96

List of Abbreviations

OML	Online Machine Learning
AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
ANN	Artificial Neural Networks
MLP	Multi-Layer Perceptron
CNN	Convolutional Neural Networks
FFNN	Feed-Forward Neural Networks
CNN	Convolutional Neural Networks
RNN	Recurrent Neural Networks
MSE	Mean Squared Error
GD	Gradient Descent
BGD	Batch Gradient Descent
SGD	Stochastic Gradient Descent
MBGD	Mini-Batch Gradient Descent
ADAM	Adaptive Moment Estimation
DML	Distributed Machine Learning
GPU	Graphics Processing Units
TPU	Tensor Processing Units
IID	Independent and Identically Distributed
FL	Federated Learning
HFL	Horizontal Federated Learning
VFL	Vertical Federated Learning
FedAvg	Federated Averaging
GM	Geometric Monitoring
FGM	Functional Geometric Monitoring
FDA	Functional Dynamic Averaging
RTC	Round Termination Condition
API	Application Programmable Interface
IPC	Inter Process Communication
OSyncSGD	Online Synchronous Stochastic Gradient Descent

Chapter 1

Introduction

In recent years, the rapid development of edge computing and the Internet of Things (IoT) has led to an exponential increase in the volume of data generated by various devices. According to a recent study by IDC, by 2025, the global data sphere will grow to 175 zettabytes, with a significant portion of this data being generated by edge devices such as smartphones, sensors, and other IoT gadgets[1]. These edge devices are capable of collecting and processing large amounts of data locally, creating new opportunities for Machine Learning (ML) applications that leverage this data to develop more intelligent and responsive systems[2].

Traditional ML approaches often rely on centralized data collection and processing, where data from multiple sources is aggregated into a central server for training models[3]. While effective, this approach poses significant challenges, particularly concerning data privacy and security. Centralized data storage can be vulnerable to breaches and unauthorized access, raising concerns about the confidentiality of sensitive information[4]. In response to these concerns, stringent data protection regulations such as the General Data Protection Regulation (GDPR) in the European Union and the California Consumer Privacy Act (CCPA) in the United States have been enacted[5, 6]. These laws mandate strict guidelines for data handling and emphasize the need for privacy-preserving techniques. Additionally, transferring vast amounts of data to a central server incurs substantial communication costs and latency issues. Not to mention the enormous computational resource requirements for a sole system to handle the tremendous amount of data gathered[7].

Federated Learning (FL) has emerged as a promising solution to address these challenges. FL allows decentralized training of ML models by enabling edge devices to collaboratively learn a shared model while keeping the data localized[8]. This approach significantly enhances data privacy and security, as raw data never leaves the device while distributing the computational load. Moreover, FL reduces communication overhead by transmitting only model updates instead of raw data, making it more efficient for large-scale deployments[9].

Additionally, FL systems can operate in an online manner, continuously processing and updating models as new data arrives[10]. This is crucial for environments where data is ceaselessly generated, such as IoT settings, allowing FL to adapt in real time without the need for traditional learning methods. This real-time capability ensures that models remain up-to-date with the latest data, enhancing their relevance and accuracy in dynamic environments[11].

In summary, the Federated Learning procedure involves edge devices training local models on their individual datasets and then sending only the model updates to a central server. The server aggregates these updates to form a global model, which is then redistributed back to the edge devices, thus iterating the process until a terminating condition is met.

Technological colossus Google was one of the first to identify the potential of the Federated Learning (FL) approach, significantly contributing to its research and development. Recognizing the limitations of traditional centralized machine learning methods, Google pioneered FL to enhance data privacy, security, and communication efficiency. Through the development of ad-

vanced algorithms such as Federated Averaging, open-source frameworks like TensorFlow Federated, and efficient protocols for distributed communication like gRPC, Google paved the way for efficient federated training and synchronization across diverse and heterogeneous edge devices[12]. By facilitating practical implementations and promoting widespread adoption, Google's contributions have been instrumental in establishing FL as a viable and transformative approach in the machine learning landscape[8].

1.1 Gaps And Motivation

Despite the significant advancements and promises of Federated Learning (FL), several critical challenges must be addressed to realize its full potential. While FL offers substantial benefits in terms of data privacy, security, and efficient decentralized model training, there are notable gaps in current research and practical implementation that warrant attention. The most prevalent gaps in FL research need to be highlighted before delving into a thorough analysis.

To begin with, the deployment of FL often revolves around intricate architectures that require a meticulous and profound understanding of distributed systems operations[13]. Unlike centralized systems, where data is aggregated and processed in a single location with highly refined deep learning algorithms—the culmination of years of research—FL necessitates a sophisticated distributed framework capable of managing data across numerous edge devices[2]. This inherent complexity of FL complicates the transition from centralized setups to decentralized alternatives, as well as the integration of existing machine learning frameworks and algorithms, due to the overhead involved in the design and implementation processes.

Even when the complexities of designing and implementing the system are addressed, the evaluation and testing of FL applications are typically conducted under idealized conditions[14]. These conditions do not capture real-world challenges, such as fluctuating bandwidth, latency, and device mobility. Such controlled testing environments fail to reflect the true performance and robustness of FL systems in dynamic and unpredictable settings[15]. One major contributing factor is the absence of frameworks for FL that provide built-in network simulation support. This issue is exacerbated by the fact that most testing and evaluations of FL systems are conducted in laboratory networks with stable and high-speed connections. Consequently, models that perform well in simulations may encounter unforeseen issues when deployed in actual environments, as edge devices often operate in dynamic and unpredictable conditions, leading to suboptimal performance and reduced reliability.

Moreover, while there are frameworks available for data management in FL, such as TensorFlow Federated, they are typically restricted to simulation setups and do not adequately support real-world deployments. This restriction to simulation environments means that potential discrepancies between simulated performance and real-world performance are further magnified. This limitation hinders the validation and optimization of FL systems, as developers and researchers cannot fully evaluate the practical viability and scalability of their solutions in real-world scenarios. This gap limits the potential for FL to be effectively integrated into applications where the benefits of improved privacy, reduced communication costs, and efficient distributed learning are most needed[13].

Last but not least, there is a pressing need for optimized and effective synchronization techniques in the realm of Online Machine Learning (OML). These techniques must balance the communication overhead, which can become a substantial issue in large-scale FL deployments[16]. The communication-to-computation ratio in FL systems can indeed become unfavorable. In distributed learning, the frequent communication of model updates between edge devices and the central server can lead to significant communication overhead. This issue is particularly pronounced

in FL environments, where edge devices often have heterogeneous computational and communication capabilities[17]. The disparity in these capabilities further complicates the synchronization process, leading to uneven training progress and reduced overall system efficiency[18]. This is especially crucial in the current era, often referred to as the Big Data Era, where the volume of data being produced is constantly increasing and the scope of FL deployments is becoming substantial.

1.2 Scientific Contributions and Employed Approach

This thesis aims to address the aforementioned challenges with a comprehensive and holistic approach. The primary objective is to introduce an all-encompassing Federated Learning (FL) framework that bridges existing infrastructure gaps. This research presents a flexible FL architecture that integrates the Flower Framework for FL orchestration with the NS3 Simulator for network condition management. This integration aims to emulate realistic conditions to assess the efficiency of the implemented Geometric Monitoring approach of Functional Dynamic Averaging (FDA). The robustness of the system and the effectiveness of the synchronization technique are rigorously evaluated throughout this study.

To achieve these objectives, a detailed and modular approach was employed, necessitating the seamless cooperation of three distinct components: the Federated Learning orchestrating framework, the optimized synchronization technique, and the network simulation utility. Each element was developed independently and subsequently integrated to form a cohesive system. The integrated platform was then subjected to extensive testing to validate its resilience and the efficacy of the FDA method. The approach can be summarized as follows:

1. *Implementation of the Federated Learning Orchestrator:* The Flower framework is utilized to construct a FL network with a central server and multiple clients connected in a star topology. This setup is algorithm-agnostic and provides a versatile platform that can load and prepare any desired dataset for the federated process, handle various artificial neural network architectures (ANNs), and manage diverse aggregation techniques. Flower's high-level abstraction allows for the parameterized management of all facets of the FL setup and execution.

2. *Application of the Geometric Method for Synchronization:* The training process is customized to achieve the desired synchronization. A tailored implementation of a model is developed, with modifications to fit the FDA synchronization requirements. Additionally, a new component, the *Metric Server*, is introduced to monitor synchronization across clients, ensuring effective communication and coordination.

3. *Implementation of the Network Simulator:* The NS3 network simulator is employed to create a realistic federated learning communication environment. This simulation computes various metrics that indicate network quality, stability, and overall communication performance between clients and the server.

4. *Integration of the FL Platform with the Network Simulator:* The integration of the two modules is accomplished using a socket-based Inter-Process Communication (IPC) protocol to facilitate message transfer between the FL and NS modules. An interface is designed on each side to handle sending and receiving operations, ensuring seamless interaction between the components.

5. *Thorough Testing and Evaluation:* Simulation scenarios are devised to validate both the architecture's robustness and the synchronization logic's effectiveness. A data frame structure is used to store and manage simulation results, enabling efficient visualization and manipulation of the collected data.

1.3 Thesis Outline

The thesis research is structured into seven chapters. The first one, *Introduction*, will be excluded from the upcoming list. A brief overview of the subsequent chapters is provided below:

- *Chapter 2: Literature Review* - This chapter lays the groundwork of the theoretical background that led to this research study. It encompasses concepts from the fields of Machine Learning, Artificial Neural Networks, Optimization, Distributed Machine Learning, Federated Learning, and System Networking. The contents of these sections equip the reader with the necessary understanding of advanced terms and practices utilized in this thesis. The topics are presented in an order that transitions from general to specific, aligning closely with the thesis topic.
- *Chapter 3: Related Work* - This chapter reviews relevant works conducted in the domain of our research. It includes benchmarked datasets, ANN architectures, FL algorithms, synchronization techniques, and a past attempt to create a unified FL and network simulation framework. These works have paved the way for our implementation.
- *Chapter 4: Environment Replication* - This chapter showcases the software tools employed for the development of the thesis environment, enabling its recreation by the reader.
- *Chapter 5: Methodology* - This chapter provides a thorough description of the applied methodology with a step-by-step overview of the implementation of the researched system. It includes all implementation aspects, starting with showcasing the top-level architecture and proceeding with data preparation and distribution, model development, and Federated Learning and Network Simulation implementation and integration. It concludes with the monitored metrics for the system's evaluation.
- *Chapter 6: Experimental Results and Discussion* - This chapter presents the experimental setup, results, and analysis of the experimentation outputs. It discusses the performance and effectiveness of the implemented methodologies and frameworks.
- *Chapter 7: Conclusion and Future Work* - This chapter summarizes the findings of the research and suggests directions for future work.

Part I

Literature Review

Chapter 2

Theoretical Background

The following chapter offers a comprehensive overview of the key theoretical concepts integral to this research study, as previously outlined. It systematically presents the scientific principles amassed, to cultivate a unified understanding of prior knowledge that made this research feasible. The discussion begins with broader terminologies and progressively delves into more complex notions that will be explored in subsequent sections. This structured approach aims to facilitate a deeper understanding of the theoretical underpinnings essential to this research while highlighting the underlying connection among the various terms presented.

2.1 Fundamentals of Machine Learning

2.1.1 Bridging Machine Learning and Artificial Intelligence

In the rapidly evolving domain of computer science, Artificial Intelligence (AI) and Machine Learning (ML) exhibit a compelling interrelationship, particularly promising in an era abundant with data. As the demand for intelligent applications escalates, it becomes imperative to distinguish clearly between these two frequently conflated terms. As critically highlighted by Kühl et al.[19], the terms AI and ML are often used interchangeably both in scholarly literature and expert discussions, leading to considerable ambiguity.

Essentially, ML can be seen as a subset of AI, primarily focused on designing systems that autonomously learn from data and enhance their performance over time without explicit reprogramming. In contrast, a widely acknowledged definition of AI, as provided by Russell and Norvig[20], positions AI as "the study of agents that receive percepts from the environment and perform actions." On this basis, the authors classify AI systems into four types: Those that think like humans, think rationally, act like humans, and act rationally.

Building on this classification, it provides a fundamental framework for understanding the diverse objectives and methodologies within AI research. It emphasizes AI's ability to perceive its environment through data acquisition and respond through actions, thereby linking it closely with fields such as machine learning, robotics, natural language processing, and reasoning. This clear delineation not only helps in understanding AI's broad applications but also illustrates the foundational role of ML within the broader AI landscape.

2.1.2 The Taxonomy Machine Learning Techniques

Now that a clear distinction between the terms AI and ML is established, it is crucial to examine the nature of the diverse methodologies encompassed by ML. The generalisability of published research on this issue remains a challenge. Over recent years, various studies have offered differing classifications of ML methods, with the determining factor being the scope under

which ML was investigated. Predominantly, existing literature has been concentrated on the nature of the feedback mechanism applied, primarily categorizing ML algorithms into supervised, unsupervised and semi-supervised [21, 22] with some studies also incorporating reinforcement learning into their analysis [4, 23].

This research, however, adopts a more holistic approach, aspiring to grasp the whole spectrum of ML applications. Instead of focusing solely on the nature of the learning feedback, this classification also considers the objectives of the learning tasks and the timing of data availability. This broader taxonomy, initially introduced by Zhou et al.[7], encapsulates all the critical elements that influence the design and deployment of ML algorithms. The suggested taxonomy is the one shown below in fig. 2.1.

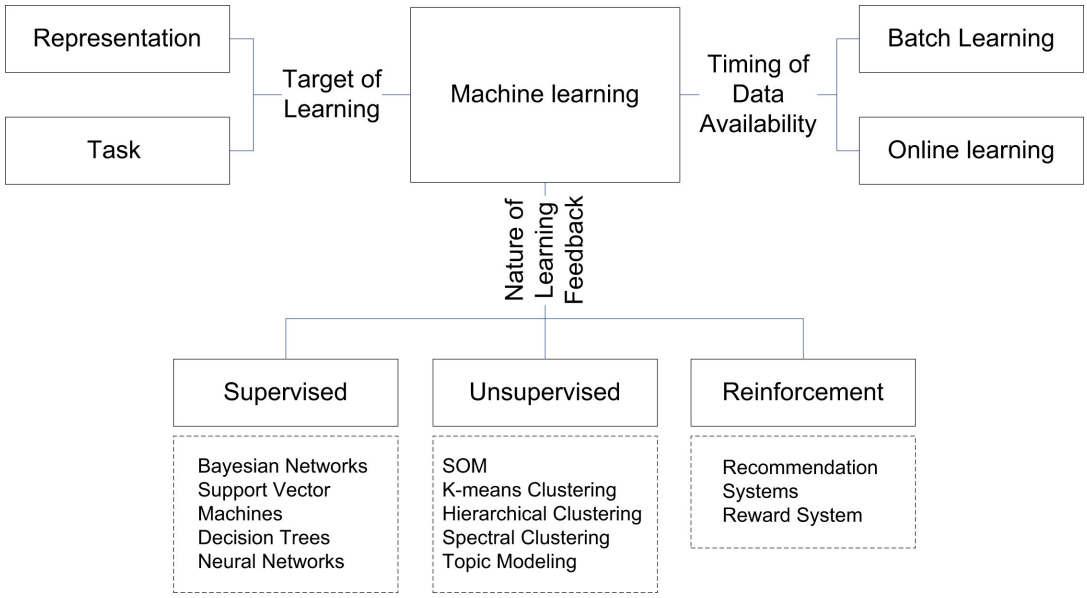


Figure 2.1. *Multi-dimensional taxonomy of Machine Learning[7]*

- **Nature of Learning Feedback:** Based on this dimension of the taxonomy applied to Machine Learning, three principal system categories and one secondary system emerge: supervised, unsupervised, and reinforcement learning systems and semi-supervised (secondary). In supervised learning, sample input-output pairs are presented to the system, which in turn attempts to infer a function able to map inputs to corresponding outputs. In practical terms, supervised ML techniques strive to construct a predictive model by applying algorithms to a set of known data points to extrapolate insights to an unseen dataset[23, 24].

  Unsupervised learning, in contrast, can be identified as a more data-driven approach, where datasets devoid of labels are analyzed. This means that while the structure of the data is apparent; no explicit guidance is provided as to the expected outcome and the aim is to uncover inherent patterns within the data without predefined answers.

  Likewise, reinforcement learning is not provided with input-output pairs, thus resembling unsupervised learning. Yet it requires a form of feedback—namely rewards or penalties linked to the actions taken. This environment-driven feedback mechanism guides the learning process towards optimal decision-making in an effort to minimize loss or maximize reward, depending on the environment's setup[25]. The distinct applications of these techniques are demonstrated by Qiu et al.[4], who assert that supervised and unsupervised learning primarily concentrates on data analysis, whereas reinforcement learning is better suited for decision-making tasks.

Additionally, semi-supervised learning which sits at the intersection of supervised and unsupervised learning, entails learning from a limited set of labeled data complemented by a larger pool of unlabeled data. The aim of this learning method is similar to supervised learning, yet the knowledge is gathered both from annotated and unannotated data. Some of the key algorithmic examples of each category are demonstrated in the following table 2.1;

Learning type	Model building	Examples
Supervised	Algorithms learn from labeled data (task-driven)	Classification, regression
Unsupervised	Algorithms learn from unlabeled data (data-driven)	Clustering, associations
Semi-supervised	Models built using combined data (labeled + unlabeled)	Classification, clustering
Reinforcement	Models based on reward or penalty (environment-driven)	Classification, control

Table 2.1. *Learning types based on feedback along with key examples of each field*

- **Target of Learning:** Machine learning can be segmented into representational and task learning based on the target of learning—whether it focuses on input features or specific tasks respectively. Representational learning seeks to identify novel data representations that simplify the extraction of useful information for constructing classifiers or other predictors[26]. This approach frequently entails isolating critical features that display firm influence upon data variability, which can be proven exceptionally practical in probabilistic models aiming to capture the posterior distributions of fundamental factors. Common processes in this category include density estimation, which attempts to define the underlying probability density of data and dimensionality reduction, aimed at reducing the complexity of data from a higher to a lower-dimensional space. Last but not least, another fundamental example of representational learning is Deep Learning (DL) which is one of the fastest-growing fields of ML providing a viable approach for big data processing. All things considered though, defining clear objectives in representational learning can be challenging.

  On the contrary, task learning is more outcome-oriented, targeting definite outputs, and can be recognized both in supervised and un-supervised setups. To offer an illustration, classification and regression, generally categorized under supervised learning, aim to predict discrete classes and continuous values respectively[27], whereas clustering-a form of unsupervised learning-groups data without predefined categories[28]. All things considered, each technique addresses specific analytical needs and applications within the broader ML framework.

- **Timing of Data Availability:** On the basis of data availability timing, Machine Learning can be classified into online and batch learning. Briefly, batch learning algorithms consolidate knowledge by training comprehensively on the entirety of the dataset available at a certain point in time, whereas online learning algorithms process data continuously in streams, accommodating updates through individual samples or mini-batches[10].

  To elucidate these methodologies, in the offline learning scenario typical of batch processing, once training is complete and the model is exhibiting adequate performance on the test set, the model is finalized and deployed. In case of new data becoming available, indicating a need for model recalibration, a new cycle of training and evaluation needs to be performed integrating both existing and new data. Another key factor to note is that the batch learning process functions under the belief that data are Independent and Identically Distributed (IID) or that data are drawn from the same probability distribution[29]. Nonetheless, this assumption often diverges from the complexities of real-world data dynamics.

Conversely, online learning operates under no statistical assumptions and adapts continuously to data, rendering it indispensable in environments where data generation is ceaseless or complete dataset training is impractical. Last but not least, another principal difference between the two ML categorizations is that batch training is expected to generalize, producing a more broad reflection of knowledge, while this principle does not uniformly apply to online learning, which focuses on immediate, accurate predictions upon incoming data[29].

All things considered, the above multi-dimensional taxonomy can be applied to any given machine learning algorithm, resulting in a categorization along the three discrete axes displayed at fig. 2.1. For instance, the researched system in this thesis can be categorized as a supervised learning system that performs classification of images, utilizes DL with Convolutional Neural Network (CNN) architectures for representational learning, and operates in an online learning environment to accommodate the continuous influx of data characteristic of the problem domain.

2.1.3 Model Complexity & Challenges

In order to measure the efficiency and generalizability of Machine Learning algorithms, two crucial terms to consider are model performance and complexity. Model performance refers to the effectiveness of an ML model in making accurate predictions or decisions based on new, unseen data[30]. Performance is typically assessed using metrics such as accuracy for classification or mean squared error for regression. For the purposes of this study, accuracy is the observed metric and can be calculated as follows:

$$accuracy = \frac{CorrectObservations}{TotalObservations} \tag{2.1}$$

In addition to quantifying efficiency, model performance provides a tangible estimator for whether a model is improving or not with new data while enabling different ML algorithm comparisons under a common scope.

Model Complexity

Model complexity relates to a model's ability to represent a wide range of functions, primarily influenced by its architecture, such as the number of parameters, the depth, and the types of functions it can incorporate. Essentially, complexity determines the model's capacity to capture intricate patterns within the data. Increasing a model's complexity means introducing more parameters, thereby enhancing its adaptability to diverse patterns. However, this increase comes at a cost: while the risk of underfitting diminishes, the danger of overfitting becomes significant, as the model might become excessively tailored to the training dataset, making it too complex for the data it needs to generalize from[31, 32].

Balancing Challenges: Overfitting and Underfitting

The key to optimizing model performance lies in carefully adjusting its complexity to find a balance between overfitting and underfitting. Overfitting happens when a model captures not only the underlying patterns but also the noise within the training data, leading to excellent performance on the training set but poor generalization to new, unseen data [31]. Conversely, underfitting occurs when the model is too simplistic, failing to capture the true patterns in the data, which results in suboptimal performance on both the training and test datasets. This issue often arises when the model lacks sufficient complexity to learn the relationships between input features and the target variable.

Techniques to Achieve Generalizability

To find the optimal complexity where the model effectively extracts valuable patterns from the data without overfitting, several techniques are commonly employed:

- *Regularization:* This involves adding a penalty term to the loss function to prevent the model from becoming overly complex. Techniques like L1 (Lasso) and L2 (Ridge) regularization are widely used to achieve this.
- *Cross-validation:* This technique evaluates model performance on unseen data by partitioning the dataset into multiple training and validation sets, and averaging the evaluation metrics across all partitions to ensure robust performance.
- *Early stopping:* This involves monitoring the model's performance on a validation set during training and halting the training process when the performance on the validation set starts to decline, thereby preventing overfitting.

These strategies are essential for maintaining the delicate balance between overfitting and underfitting, ensuring that the model remains generalizable and performs well on unseen data[33].

2.2 Artificial Neural Networks

Artificial Neural Networks (ANNs), a subset of ML models, are a pivotal research area in Machine Learning, forming the backbone of Deep Learning, which has transformed handling vast and complex datasets. ANNs mimic the human brain's architecture, consisting of layers of interconnected neurons designed for specific computational tasks [34]. ANNs have significantly advanced fields such as image and speech recognition, natural language processing, and decision-making under uncertainty. Their versatility extends to various applications, including healthcare diagnostics, autonomous vehicles, and financial modeling, demonstrating robustness and adaptability[35]. This study focuses on online learning ANNs applying supervised learning for classification.

2.2.1 Perceptron and Activation Functions

Rosenblatt and Multi-Layer Perceptrons

Perceptrons, regarded as one of the earliest and simplest forms of artificial neurons, were initially introduced by Frank Rosenblatt in 1958 [36]. Perceptrons functionality is closely related to classification tasks and can be divided into two categories; Rosenblatt Perceptrons and Multi-Layer Perceptrons (MLP).

The first definition of perceptrons resembles a single-layer neural network with m input sources and one sole output neuron, where stimuli can be classified into two classes, using a linear decision boundary based on the Heaviside function. This single-layer structure, restricts Rosenblatt perceptron flexibility, limiting its functionality to linearly separable problems and binary classification tasks. To demonstrate the functionality of this kind of perceptron, a more strict mathematical description seems necessary[37]:

$$\begin{cases} v = \sum_{i=1}^{m} w_i x_i + b, \\ y = \phi(v) \end{cases} \tag{2.2}$$

Expanding on the definition provided by eq. (2.2), Consider x_i (for $i = 1, \ldots, m$) as the set of input signals delivered to a perceptron. In this model, each w_i represents the synaptic weight associated with the perceptron for the corresponding input x_i. The term b is known as the

externally set bias. The term v is referred to as the *induced local field* or simply the linear combiner of inputs and weights. The function φ, which is an activation function or a step function, processes this induced local field to produce the output y (see fig. 2.2) of the perceptron. This perceptron model essentially consists of a linear combiner capped by a step function, which decisively determines the output based on the calculated local field v.

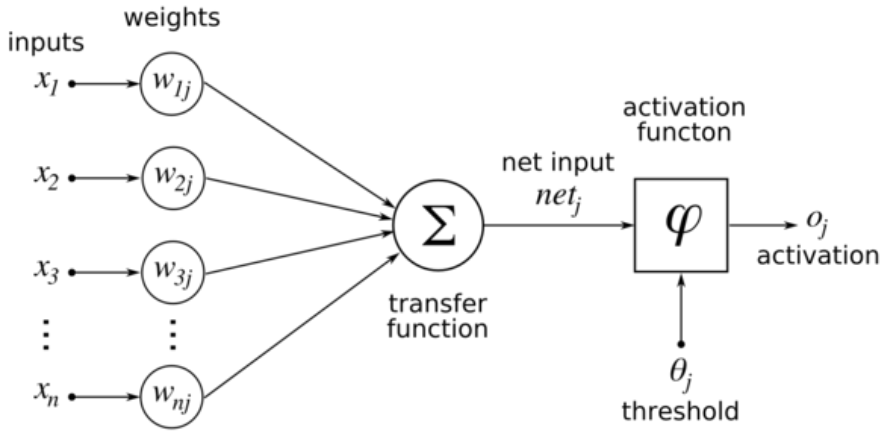


Figure 2.2. *Rosenblatt's perceptron architecture*

The activation function of Rosenblatt' perceptron, also referred to as the Heaviside step function, is a straightforward threshold function summarized by eq. (2.3). The binary output of the function allows us to determine whether an input sample belongs to one out of two classes limiting strictly in binary classification problems [38]. For this reason, more adaptable activation functions were introduced for various use cases which will be outlined later:

$$\varphi(v) = \begin{cases} 1 & \text{if } v > 0, \\ 0 & \text{if } v \leq 0. \end{cases} \tag{2.3}$$

Despite the foundational impact of Rosenblatt's perceptrons in neural networks, their limitations with non-linear problems created a need for a more sophisticated approach. This led to the development of Multi-Layer Perceptron (MLP) models, which addressed the famous XOR problem highlighted by Marvin Minsky and Seymour Papert[39]. By incorporating multiple layers of neurons, including hidden ones, MLPs can learn intricate patterns in data, enhancing their learning capabilities and handling non-linear transformations. This advancement improved the network's ability to generalize by learning hierarchical features and adapting to various levels of complexity, from linear problems to complex decision boundaries that single-layer perceptrons could not manage.

The architecture of a Multi-Layer Perceptron (MLP) is outlined as follows:

- *Input layer:* n source neurons.
- *Hidden layers:* one or several hidden layers each containing containing h_i neurons.
- *Output layer:* m output neurons.

Each neuron in the network employs a differentiable non-linear activation function. The most commonly utilized activation function in MLPs is the sigmoid which is close to linear near the origin while saturating rather quickly when getting away[38]. This allows multilayer perceptrons to model well both strongly and mildly nonlinear relations. Additionally, the structure of this MLP is fully connected, meaning every neuron in a given layer is connected to all neurons in the

subsequent layer (see fig. 2.3). Signals in the network flow unidirectionally from the input layer to the output layer, moving sequentially through each layer from left to right.

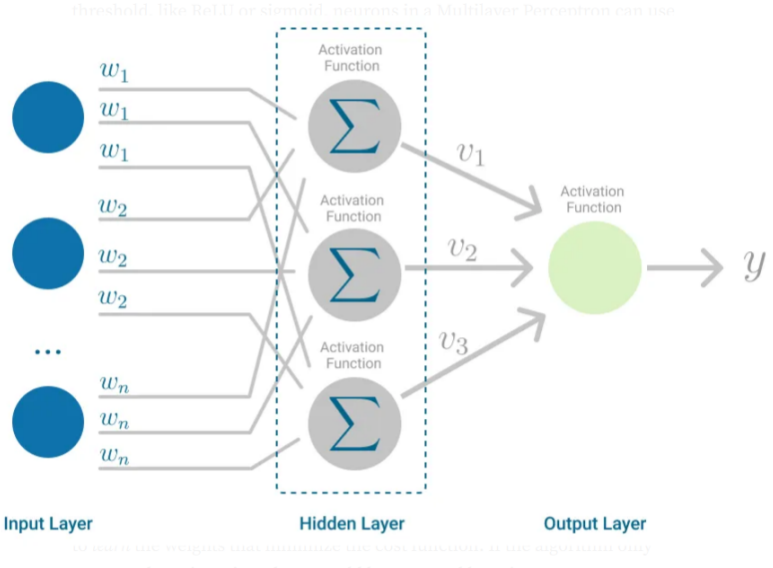


Figure 2.3. *Multi-Layer Perceptron (MLP) architecture with n input neurons, one hidden layer with $h_1 = 3$ neurons, and $m = 1$ output neuron*

Evolution of Activation Functions

The functionality of neural networks is greatly influenced by the choice of activation functions. Initially, Rosenblatt's perceptron used the Heaviside step function, a basic activation function that activates based on a simple threshold rule.

As neural network theory developed, more flexible activation functions were needed to enable learning in multi-layer architectures and address non-linear classification problems that single-layer perceptrons could not solve. The sigmoid function emerged as one of the most popular choices for Multi-Layer Perceptrons (MLPs) because of its continuous and differentiable nature, which is crucial for gradient-based learning algorithms like backpropagation.

Over the years, several other activation functions have been introduced to meet specific requirements of neural network architectures and problems. Common examples include the hyperbolic tangent (tanh), which is similar to the sigmoid but ranges from -1 to 1, centering the data; the Rectified Linear Unit (ReLU), popular in deep learning networks due to its efficiency and effectiveness at preventing the vanishing gradient problem[40]; and Softmax, predominantly used in the output layer of Convolutional Neural Networks (CNNs) to convert logits into a probability distribution over multiple exclusive classes (see section 2.2.2).

These activation functions have been extensively researched to maximize neural network performance[41, 42]. Table 2.2[1] presents a summary of commonly used activation functions.

These developments have significantly shaped the design and implementation of modern neural networks. The shift from simple threshold functions to complex non-linear functions has enabled neural networks to model intricate patterns in data, improving their ability to generalize from training data to unseen situations. This transition marks a pivotal evolution from early perceptrons to the sophisticated networks at the forefront of artificial intelligence research and applications today.

[1] The indexed v_i in Softmax's equation in the table symbolizes the reference to class i.

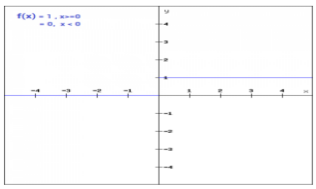
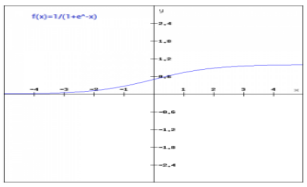
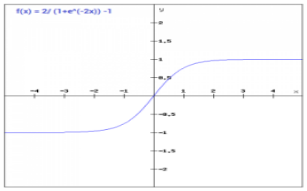
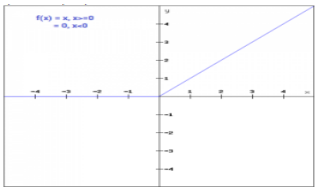
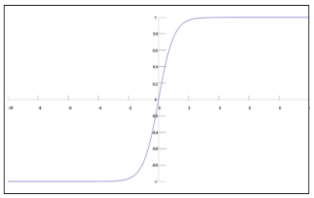
Function	Equation	Use	Graph
Heaviside Step	$\varphi(v) = \begin{cases} 1 & \text{if } v > 0 \\ 0 & \text{if } v \leq 0 \end{cases}$	Binary Classification	
Sigmoid	$\varphi(v) = \frac{1}{1+e^{-v}}$	Probabilities	
Hyperbolic Tangent (tanh)	$\varphi(v) = \frac{e^{v}-e^{-v}}{e^{v}+e^{-v}}$	Centered Data	
Rectified Linear Unit (ReLU)	$\varphi(v) = \begin{cases} v & \text{if } v > 0 \\ 0 & \text{if } v \leq 0 \end{cases}$	Sparse Activation	
Softmax	$\varphi(v_i) = \frac{e^{v_i}}{\sum_k e^{v_k}}$	Multiclass Classification	

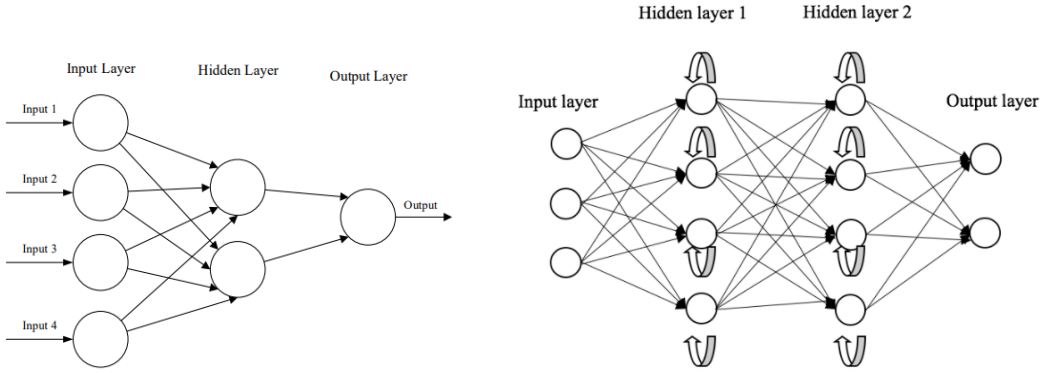
Table 2.2. *Commonly employed activation functions in neural networks[42]*

2.2.2 Network Architectures

The research in the domain of neural networks always strives for optimization, and to be precise, maximization of performance and generalization[43]. Hence, in an effort to expand the limits of what can be achieved with Neural Networks, several diverse architectures have been proposed, tailored for specific data and fields. That way performance is improved by addressing the specific needs of a given problem. While the types of architectures developed to this day are countless, this analysis will be restricted to a brief overview of two major categories and afterward, there will be a shift in focus toward the architecture most valuable for our research.

First and foremost, there are Feed-Forward Neural Networks (FFNN)[46] which represent the simplest form of neural networks, designed on the tracks of MLPs' logic discussed previously (see fig. 2.4a) [2]. In this type of networks, neurons are organized in layers with each neuron being connected to all neurons of subsequent layers. The network's flow is unidirectional from the input layer through hidden layers to the output layer. Common use examples encompass image and speech recognition, due to their simplicity and effectiveness.

[2] FFNNs are a superset of MLPs. They additionally contain single-layer and more complex architectures without circles.



(a) *FFNN with one hidden layer[44]* **(b)** *General structure of RNNs[45]*

Figure 2.4. *Side by side illustration of FFNN and RNN architectures*

Moving forward, another crucial type of neural network architecture is Recurrent Neural Networks (RNNs)[45]. Recurrent Neural Networks are designed to handle sequential data[47]. Differing from traditional neural networks, RNNs process inputs using a loop mechanism that allows information from previous steps to influence current and future operations (see fig. 2.4b). This architecture [44] makes RNNs ideal for applications such as language modeling and text generation, where the sequence and context of information are critical. They are particularly effective in tasks like speech recognition, music generation, and other forms of sequential pattern recognition.

Convolutional Neural Networks (CNNs)

Building on the foundational principles of Feed-Forward Neural Networks, the exploration into network architectures takes a sophisticated turn with Convolutional Neural Networks (CNNs)[48], a significant achievement in Deep Learning. CNNs have broad applicability in fields such as image and video recognition, recommendation systems, image classification, image segmentation, natural language processing, and computer vision. The name of CNNs stems from the *convolution*; A mathematical operation described in calculus that combines two functions to produce a third, which expresses how the shape of one is modified by the other. This inherent property of convolutional layers provides CNNs with translation invariance, enabling them to detect and extract patterns and features from data regardless of position, orientation, scale, or translation.

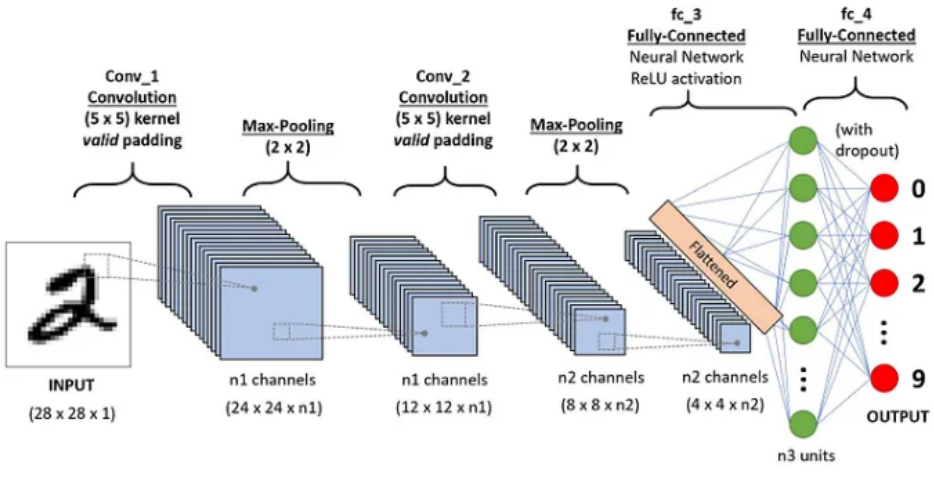


Figure 2.5. *CNN processing for handwritten digits classification[49]*

To further elucidate the methodology of a CNN, we will observe the example of fig. 2.5 depicting a single input from the MNIST dataset[3] while being processed. The format in which the system perceives the input is a pixel representation of the image. After the input is received the sample undergoes a series of critical operations that eventually lead to the system's output.

The *convolution layer* performs a sliding window function to the matrix of pixels representing the digit's image, commonly known as kernel or filter. Several filters of equal size are applied, and each filter is used to recognize a specific pattern from the image, generating multiple feature maps. Observing fig. 2.6, a 3(*height*)*x*3(*breadth*) kernel is utilized, sliding over the input image, calculating the convolution result at each point and populating the feature map.

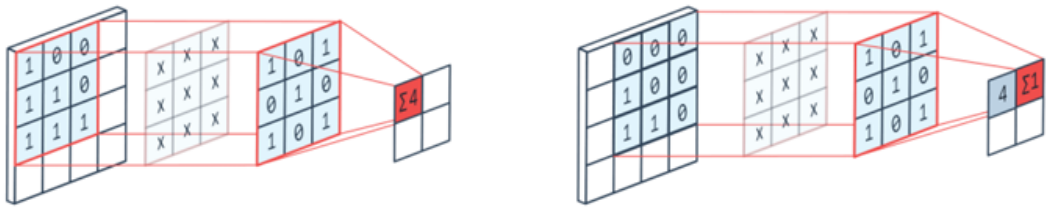


Figure 2.6. *Convolution of a* 4×4 *image with a* 3×3 *kernel highlighting the sliding window operation[50]*

After each convolution operation, a ReLU (see table 2.2) activation function is applied to deduce non-linear relationships between the features in the image, hence improving the network's capability for capturing complex patterns.

Following the convolution and activation steps, the *pooling (or subsampling) layer* reduces the spatial dimensions of the feature maps. This step is crucial as it decreases computational requirements and the number of parameters, helping prevent overfitting. Max pooling, which selects the maximum value from each patch of the feature map, is the most frequently used method (fig. 2.7a). Another common method is average pooling, which calculates the average value of each patch (fig. 2.7b). The final pooling layer converts the feature map into a flattened, one-dimensional array, preparing it for input into the fully connected layer.

(a) *Max pooling* **(b)** *Average pooling*

Figure 2.7. *Demosntration of Max and Average pooling operations*

The *fully connected layers* represent the last layer of a CNN and their inputs correspond to the flattened one-dimensional matrix generated by the last pooling layer. ReLU activations functions are applied to them for non-linearity again (as shown in fig. 2.5). Finally, a softmax prediction layer is used to generate probability values for each of the possible output labels, and the final label predicted is the one with the highest probability score.

[3]Dataset of handwritten digits

2.2.3 Learning in Neural Networks

Having established the principal components and architecture of Artificial Neural Networks (ANNs), we now proceed with a brief demonstration of their learning method. Learning in neural networks, which is also called training or fitting in ML terminology, involves the adjustment of the network's internal parameters to minimize the discrepancy between the actual output and the target output. This process is crucial for developing models that can perform well on unseen data. To successfully grasp the reasoning behind the training process, it is crucial to decompose the procedure into smaller, comprehensive sub-processes[51]. This analysis partitions the training process into six parts as follows:

1. *Initialization:* The initialization of weights is a critical first step in the training of neural networks. Proper initialization sets the stage for an efficient and effective learning process, influencing how quickly a network converges to a solution, if at all. Initially, weights can be set randomly, but more sophisticated methods such as He initialization or Xavier initialization are commonly used to improve the convergence properties of deep networks[52]. These methods adjust the scale of the weights according to the number of input and output neurons, promoting an even distribution of activations across layers during the initial phases of training[53].

2. *Forward Propagation:* This step involves computing the output of a neural network, starting by feeding the input data and calculating the weighted sum of each neuron's inputs. The activation or non-activation of the neuron is determined by the output of the nonlinear activation function (see table 2.2) over the calculated sum. The process begins at the input layer and proceeds sequentially through the hidden layers and finally to the output layer, where a prediction is made. The specific transformations applied to the neurons' sums depend on the network's architecture (e.g., convolutional, recurrent) and the types of layers used[54].

3. *Loss Computation:* A cost (or loss) function, quantifies the error between the predicted outputs and the actual target values. The cost function selection can significantly affect the network's performance and convergence. Common choices include Mean Squared Error (MSE) for regression tasks and Cross-Entropy Loss for classification tasks. The cost function provides a performance measure used during backpropagation to adjust the model's weights.

4. *Back Propagation:* Backpropagation is the fundamental algorithm for training artificial neural networks. It calculates the gradients of the loss function with respect to the weights of the network by applying the chain rule of calculus. These gradients indicate the necessary adjustments to the weights to minimize the loss function and improve model accuracy. The process involves propagating the error backward through the network, updating the weights to reduce the prediction error for subsequent iterations[55].

5. *Gradient Descent:* The gradients calculated during the *back propagation phase* comprise a clear indication of the steepest ascent on the loss curve, meaning the direction of the maximizing loss. On the contrary, as the objective is to minimize the loss, it becomes obvious that the required update of the network's parameters needs to be performed moving in the opposite direction of the gradients. By iteratively adjusting the weights in this direction, the accuracy of the system increases, while The magnitude of this update is controlled by a learning rate hyperparameter of the optimization algorithm. Additional information for this matter will be presented in the following section 2.3.

6. *Iteration:* Steps 2 – 5 are repeated for each batch of training data for multiple epochs[4] until a breaking condition is satisfied.

[4] Epoch is a complete pass over the training dataset

The above process does not necessarily guarantee generalization. The above statement loops back to a previously reported issue in machine-learning-overfitting (see section 2.1.3). Therefore, training can include a validation step where the model is tested on unseen data to adjust hyperparameters and prevent overfitting.

In closing analysis, it should be noted that neural networks accumulate knowledge by iteratively adjusting their parameters based on the applied optimization algorithm. This recurring process requires performing a vast number of operations at each training step. While such operations are manageable for small-scale models, they become significantly more demanding for substantially larger networks. The computational demands for training large neural networks can become enormous.

To address these demands, training neural networks is typically performed on powerful hardware or specialized hardware accelerators, such as Graphics Processing Units (GPUs) or Tensor Processing Units (TPUs). These specialized devices are designed to handle the high computational load, thereby speeding up the training process and making it feasible to work with large and complex models[51].

2.3 Optimization Techniques

Optimization is, undoubtedly, a crucial matter in the field of Data Science and Machine Learning. This section explores various techniques stemming from optimization theory, aiming to enhance the performance of machine learning algorithms through the effective minimization of loss functions. Starting from the fundamental principles of Gradient Descent, and continuing by presenting the key variations relevant to this research approach, this analysis clarifies the role of optimizers in contemporary neural networks.

2.3.1 Gradient Descent And Its Variants

Principle of Gradient Descent

Gradient Descent and its Variants are classified as iterative optimization algorithms, based on the idea of iteratively adjusting the parameters (coefficients) of a model in the direction of the steepest descent with respect to the machine learning algorithm's cost function[56]. Essentially, the objective of the algorithm is to find the parameter values that minimize the cost function of the neural network. For this to be achieved, the gradient of the loss function at the current point is calculated, indicating how small changes in the weights affect its output. Thus, the parameters are iteratively adjusted in the opposite direction of the gradient—towards the minima point of the cost curve (see fig. 2.8).

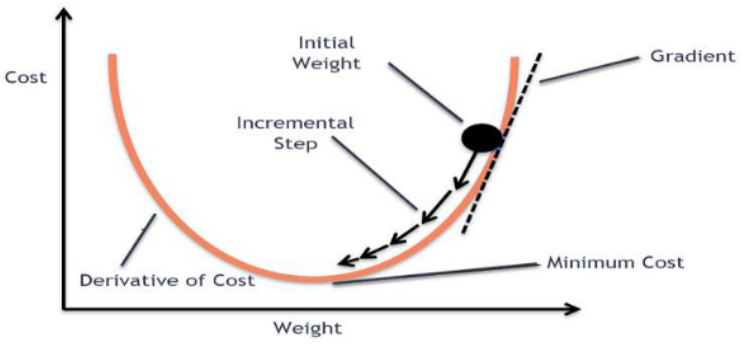


Figure 2.8. *Gradient Descent Algorithm[57]*

Batch Gradient Descent

Batch Gradient Descent (BGD), also referred to as Vanilla, computes the cost function with regard to the parameters ∂ for the entire training dataset:

$$\partial_{new} = \partial_{old} - \eta \cdot \nabla_{\partial} J(\partial) \tag{2.4}$$

Here, ∂ represents the parameters of the function to be minimized, $J(\partial)$ is the loss function, $\nabla_{\partial} J(\partial)$ denotes the gradient of the loss function with respect to ∂, and η is the learning rate, which controls the size of the incremental step taken towards the minimum.

Despite being easy to comprehend, Batch Gradient Descent (BGD) requires computing gradients for the entire dataset to update the model once, making it inefficient for very large datasets or those exceeding memory capacity[56]. Additionally, this method does not support online updates with new data as it processes the entire dataset in one go. However, batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces

Stochastic Gradient Descent

Proceeding with Stochastic Gradient Descent (SGD), this optimization method updates parameters incrementally for each training sample, $x^{(i)}$ and label $y^{(i)}$, unlike the previous discussion regarding BGD.

$$\partial_{new} = \partial_{old} - \eta \cdot \nabla_{\partial} J(\partial; x^{(i)}; y^{(i)}) \tag{2.5}$$

The term stochastic in the name refers to the random selection of one sample for the gradient calculation instead of the entirety of the dataset. This approach has been proven particularly beneficial for large-scale datasets and online learning algorithms as it aids in reducing training time. Other commonly discussed properties of SGD include the ability to effectively escape local minima, improved model generalizability to unseen data, and the detailed rate of improvement due to the frequency of updates. At the same time, this per-sample frequency of parameter updates also leads to fluctuations in the error rate, instead of a steady and controlled decrease.

Mini-Batch Gradient Descent

This approach of optimization fills the middle space between BGD and SGD, as Mini-Batch Gradient Descent (MBGD) strikes a balance between the computational efficiency of the former and the stability of the latter. Instead of updating parameters using all or only one of the training examples, Mini-Batch Gradient Descent uses a subset of n training data to perform each update:

$$\partial_{new} = \partial_{old} - \eta \cdot \nabla_{\partial} J(\partial; x^{(i:i+n)}; y^{(i:i+n)}) \tag{2.6}$$

This approach reduces the variance of the parameter updates, leading to more stable convergence than Stochastic Gradient Descent (SGD), while being significantly faster than using the full dataset as in Batch Gradient Descent (BGD). Mini-Batch Gradient Descent (MBGD) is particularly effective for training on large datasets, as it leverages highly optimized matrix operations common in state-of-the-art deep learning libraries. This results in great efficiency when computing gradients. These advantages have made MBGD the preferred optimization method when training neural networks. In fact, even when SGD is reported as the optimization algorithm, it often implies the use of mini-batches, highlighting the widespread adoption of this approach in practical applications.

2.3.2 Towards ADAM Optimizer

Learning Rate Selection

The learning rate hyperparameter, often denoted as η, is undoubtedly one of the most crucial factors contributing to the convergence of gradient-based optimization algorithms[56]. It determines the incremental step size of the algorithm towards the minimum of the cost function, as observed in fig. 2.8. The importance of balancing the hyperparameter's value cannot be overstated; too high a learning rate can cause the algorithm to overshoot the minimum, while too low a learning rate can result in excessively slow convergence or getting stuck in local minima (see fig. 2.9). Therefore, achieving balance in this situation suggests swift convergence to the minimum.

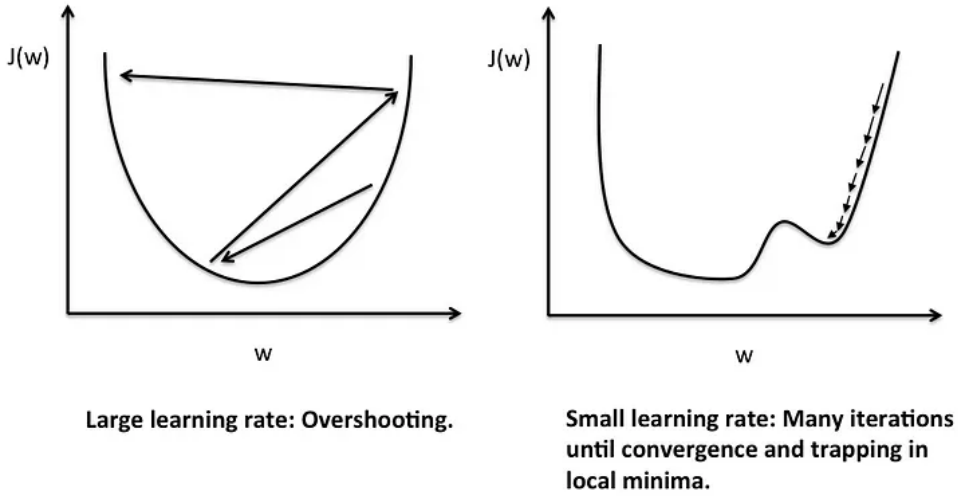


Figure 2.9. *Impact of unsuccessful learning rate selection[58]*

Momentum

Momentum is an enhancement of Gradient Descent that accelerates convergence, particularly in the context of deep neural networks. The term finds great applicability in situations where the network is not well-conditioned, leading to the composition of *ravines* —surfaces that curve more steeply in one direction than in another direction[57, 59]. Such surfaces negatively impact convergence, due to the simple fact that the gradient does not point towards the minimum, and successive steps of gradient descent can oscillate from one side to the other, progressing, albeit very slowly, to the minimum. Figure 2.10 illustrates how the addition of momentum aids in speeding up convergence to the minimum by damping these oscillations.

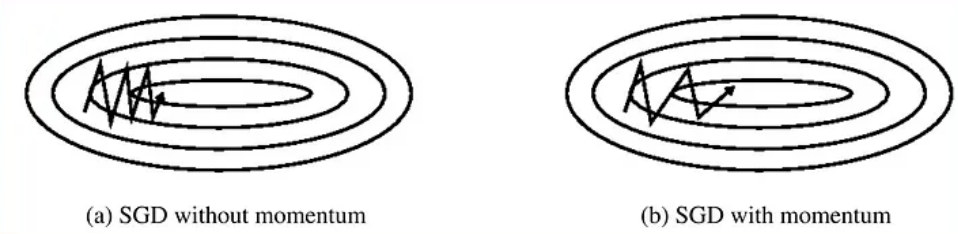


Figure 2.10. *Trajectories in ravine surfaces, with and without momentum[60]*

In the formal mathematical representation of SGD Momentum in eq. (2.7), v_t represents the velocity, β is the momentum coefficient (typically set between 0.9 and 0.99), and $\nabla_\theta J(\theta; x^{(i)}; y^{(i)})$ is

the gradient of the loss function. In other words, the velocity that is calculated for a given moment t complements the gradient descent calculation by taking into account the momentum βv_{t-1}. This way, the update on the network's parameters is computed utilizing the velocity term that includes both gradient and momentum.

$$\begin{cases} v_t = \underbrace{\beta v_{t-1}}_{\text{momentum}} - \underbrace{\eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})}_{\text{gradient descent}} & \triangleright \text{ Velocity computation} \\ \theta_t = \theta_{t-1} + v_t & \triangleright \text{ Parameters' update} \end{cases} \tag{2.7}$$

ADAM Optimizer

The ADAM (Adaptive Moment Estimation) optimizer, proposed by Kingma and Ba (2017)[61], is a widely utilized algorithm that combines the benefits of both momentum and adaptive learning rates, classifying it as an adaptive learning algorithm. This approach leverages concepts from Adagrad[62] and RMSprop[63], making it particularly effective for deep neural network (DNN) training by managing sparse gradients and non-stationary objectives.

By maintaining exponentially decaying averages, ADAM calculates adaptive learning rates of past gradients and squared gradients. These moving averages smooth out noise in the gradient updates, providing a stable convergence path. The momentum aspect, derived from the average of past gradients, helps accelerate convergence by maintaining consistent update directions, thus reducing oscillations. Its implementation is summarized by algorithm 2.1:

In the following steps, m_t and v_t represent the first moment (mean) and second moment (uncentered variance) of the gradients, respectively. The corrected estimates, $\hat{m}_t$ and $\hat{v}_t$, are used to compute the parameter updates. The decay rates β_1 and β_2 are typically set to 0.9 and 0.999, respectively, with ϵ preventing division by zero.

ADAM's ability to adaptively adjust learning rates for each parameter, based on the mean and variance of gradients, ensures robust performance across various machine learning applications. This is especially beneficial for hyperparameter tuning and managing high-dimensional loss landscapes.

ALGORITHM 2.1: *ADAM optimizer algorithm*

Require: a: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector

Initialize:
$m_0 \leftarrow 0$ ▷ Initialize 1st moment vector
$v_0 \leftarrow 0$ ▷ Initialize 2nd moment vector
$t \leftarrow 0$ ▷ Initialize timestep

while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ ▷ Get gradients w.r.t. stochastic objective at timestep t
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ ▷ Update biased first moment estimate
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ ▷ Update biased second raw moment estimate
$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ ▷ Compute bias-corrected first moment estimate
$\hat{v}t \leftarrow v_t / (1 - \beta_2^t)$ ▷ Compute bias-corrected second raw moment estimate
$\theta_t \leftarrow \theta t - 1 - a \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ ▷ Update parameters
end while
return θ_t ▷ Resulting parameters

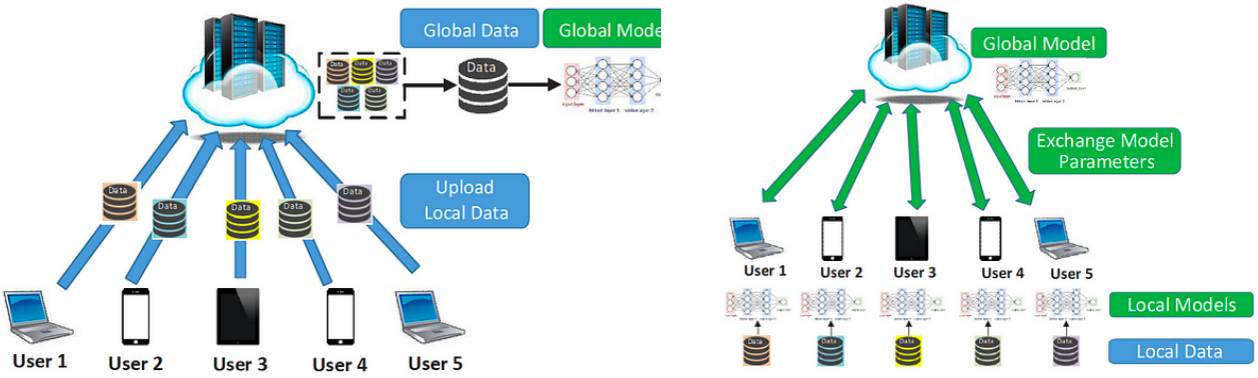
In the realm of Federated Learning, ADAM's optimization techniques are crucial for enhancing convergence rates and improving model performance. The decentralized nature of Federated Learning introduces challenges such as heterogeneous data distributions and communication constraints, which will be discussed in subsequent sections. ADAM's adaptive learning rate mechanism facilitates effective training across distributed nodes, helping to overcome data variability and ensuring efficient global model convergence. This makes ADAM particularly well-suited for the unique demands of Federated Learning, where maintaining robust and efficient training processes is essential.

2.4 Distributed Machine Learning

2.4.1 The Need for Distribution

As the field of machine learning (ML) continues to evolve, the demand for processing vast amounts of data and training increasingly complex models has surged. Traditional machine learning methodologies, which typically operate on single-machine setups, face significant limitations in terms of computational power and memory capacity. This has led to the development and adoption of centralized learning approaches, where data from various sources is consolidated into a central location for processing. While centralized learning alleviates some of the constraints of single-machine learning, it still struggles with issues such as data transfer overhead, security risks, and the inability to scale efficiently with ever-growing datasets.

To overcome these shortcomings, the paradigm has shifted towards Distributed Machine Learning (DML). DML allows for the distribution of data and computational tasks across multiple machines or devices, leveraging their collective resources. This approach not only enhances computational efficiency through parallelism, but also enables the processing of larger datasets that would be infeasible for centralized systems. By distributing the workload, DML restricts data transfer bottlenecks, enhances fault tolerance, and provides greater scalability, making it an essential advancement for modern machine learning applications.



(a) *Centralized Learning* **(b)** *Distributed Learning*

Figure 2.11. *Comparison between centralized and distributed learning approaches*

Thus, the progression from traditional machine learning (ML) to centralized learning, and finally to distributed machine learning (DML), reflects the ongoing need to optimize and scale machine learning processes to keep pace with the increasing data demands and computational complexities of contemporary applications.

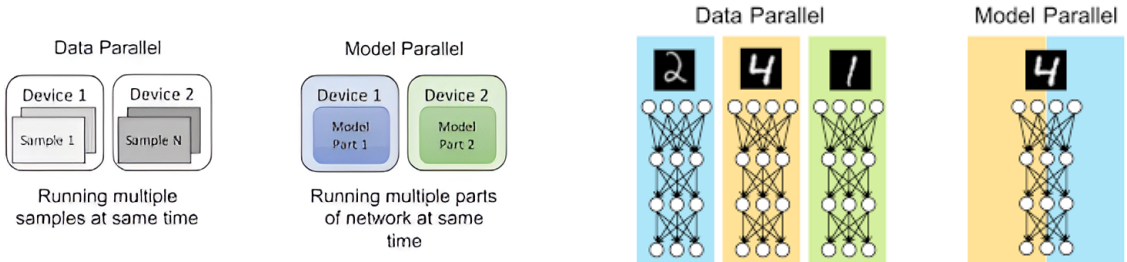
2.4.2 Distributed Machine Learning Architecture

The architecture of Distributed Machine Learning can be defined by a triad of layers dependent on each other: *The Distribution Method (Parallelism)*, *the System's Communication Topology*, and *the ML algorithm*. A broad overview of the above will be presented as follows:

Parallelism in Distributed Learning Systems

Parallelism in DML Systems can be achieved with two methods; Data and Model Parallelism. A thorough survey by Joost Verbraeken et al.[64] clarifies:

- *Data Parallelism:* In the data-parallel approach, the dataset is divided into segments corresponding to the number of worker nodes in the system. Each worker node then processes its respective data segment using the same algorithm. The model is either centralized or replicated across all worker nodes, ensuring a unified output. This method is applicable to any machine learning algorithm that assumes independent and identical distribution (i.i.d.) of data samples, which includes the majority of ML algorithms. It is also applied in our work.
- *Model Parallelism:* Model parallelism involves splitting the model itself across multiple machines and training different parts of the model on different devices. This approach is useful when the model is too large to fit in the memory of a single machine, or when certain parts of the model require more computation than others. Model parallelism is a bit more complex to implement and is less common than data parallelism, but is still used in some specialized applications, as it cannot automatically be applied to every machine learning algorithm because the model parameters generally cannot be split up.



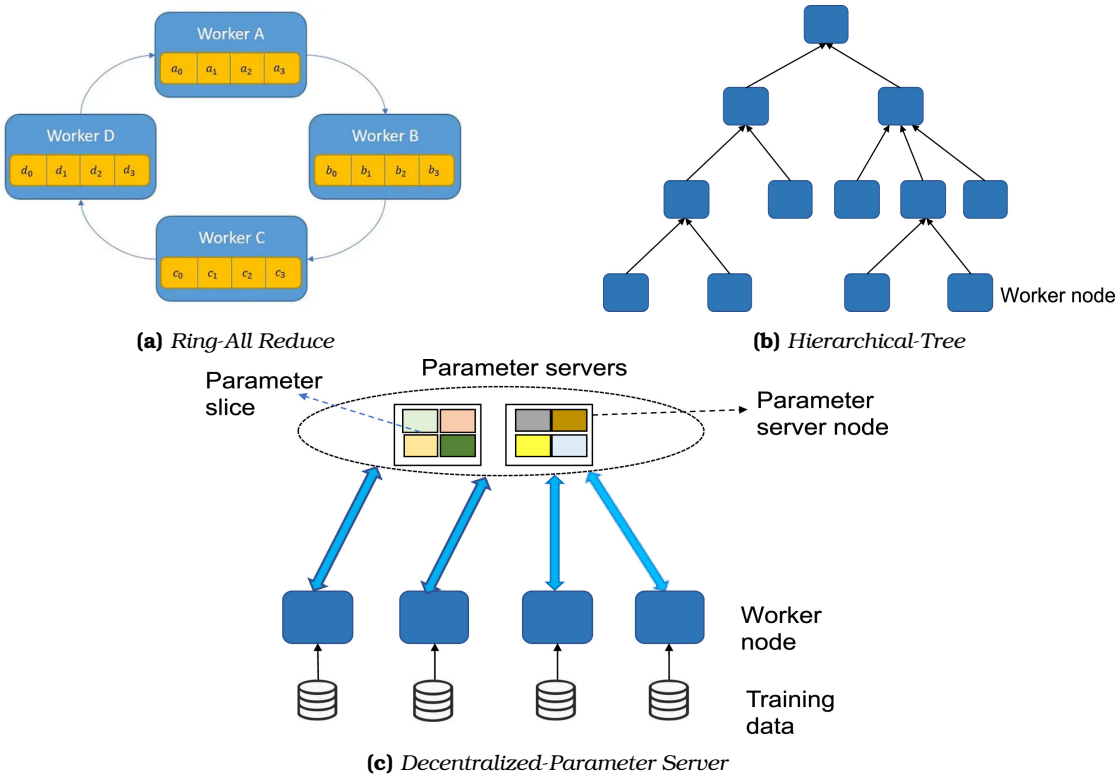
(a) *Data and Model Parallelism approaches* **(b)** *Data and Model Parallelism examples*

Figure 2.12. *Data and Model Parallelism methods*

Distributed Systems Communication Topologies

In Distributed Machine Learning (DML), the communication architecture defines the specific roles of training nodes. These roles include computation nodes, which compute gradients, and parameter aggregation nodes, responsible for aggregating these gradients and updating the model parameters. Additionally, the architecture outlines the logical topologies for node communication, such as star or ring configurations. Regardless of the model type and learning objective, this communication architecture facilitates parameter exchange and synchronization when using gradient-based optimization algorithms. A recent article by Liu et al.[2] showcases the three primary connectivity topologies used in distributed systems.

- *Ring-All Reduce:* The Ring architecture arranges nodes in a ring topology, where each node communicates directly with its two neighbors. Gradients are shared in a circular manner, with each node sending and receiving data from its adjacent nodes. This architecture efficiently



(a) *Ring-All Reduce*

(b) *Hierarchical-Tree*

(c) *Decentralized-Parameter Server*

Figure 2.13. *Principle Distributed System communication topologies*

utilizes bandwidth by limiting communication to neighboring nodes and avoids the bottlenecks of a centralized coordinator. It is particularly effective in GPU clusters and is widely used in distributed deep learning frameworks. A notable example of this topology is the Ring-All Reduce implementation provided by Uber's Horovod framework[65].

- *Hierarchical-Tree:* The Hierarchical-Tree architecture organizes nodes in a tree structure, where intermediate nodes aggregate gradients from their child nodes and pass the aggregated results up the tree to the root node. This reduces the communication overhead compared to a fully connected topology and scales efficiently with the number of nodes. The root node eventually computes the global model update and propagates it back down the tree. This architecture provides a balance between scalability and communication efficiency[66].

- *Decentralized-Parameter Server:* The Decentralized-Parameter Server architecture involves multiple parameter servers distributing the management of model parameters. Each worker node communicates with one or more parameter servers to read and update parameters during training. This decentralization reduces mitigates bottlenecks and improves fault tolerance compared to a single centralized server. The architecture requires synchronization between parameter servers to ensure model consistency across the network.

  This connectivity architecture resembles most the research's implementation, yet is restricted to a singular server to mitigate cross-server communication challenges[5]. In this scenario, all worker nodes directly communicate with the central server, simplifying network management and reducing latency in communication by having a single aggregation point. However, this also introduces a potential bottleneck at the central server and may suffer from single points of failure.

[5]Parameter server implementation with a singular server resembles a star architecture-all worker nodes directly connected to server

Distributed Learning Algorithms

Last but not least, The DML algorithm selection determines how the learning is conducted after *parallelism method* and *communication topology* are established. The objective in DML algorithms always remains to leverage parallelism and efficient communication to accelerate training and enable the handling of large datasets and models that are infeasible for single-machine setups. Therefore, to achieve their goal, data and computation loads are distributed across several nodes- either being various accelerating devices like GPUs or TPUs or completely different machines[67]. An outline of an algorithm to achieve the above is presented:

1. *Data Preparation:* Partitioning the dataset into subsets that can be distributed across multiple worker nodes, ensuring that data distribution maintains the independent and identical distribution (i.i.d.) property when required.

2. *Model Initialization:* All local models are initialized either locally by each node or centrally by a server and distributed. Depending on the needs, the parameters initialization can be random or with a specific model state.

3. *Local Training:* Each worker node processes its assigned data subset to compute local gradients. Parallelism techniques (data parallelism or model parallelism, see section 2.4.2) are applied to enhance computational efficiency.

4. *Gradient Aggregation & Synchronization:* Gradients computed by each worker are sent either to a server or a peer for aggregation, depending on the topology, to update the global model (see section 2.4.2). Synchronization can be synchronous or asynchronous, affecting the overall training dynamics.

5. *Model Update:* The updated global model is transmitted back to the worker nodes to reinstate the global parameter values to the local model.

6. *Iteration:* Steps 3 to 5 are repeated until the stopping condition is met.

At this point, we should delve deeper into the concept of *Gradient Aggregation & Synchronization* outlined previously to sufficiently cover the principal matter of synchronization in DML systems. Specifically, synchronization can be achieved either in a synchronous or asynchronous manner. Each approach inevitably comes with its own merits and challenges. To highlight these, we will briefly demonstrate the fundamental optimization algorithms for gradient calculation in both synchronization methods, namely Synchronous SGD (SSGD) and Asynchronous SGD (ASGD)

Synchronous SGD (SSGD) In SSGD, all worker nodes must complete their gradient computations and communicate these gradients before the global model parameters are updated. This ensures consistency across model updates, as all nodes work with the same model parameters in each iteration. However, this approach can lead to inefficiencies due to idle times when faster nodes wait for slower ones, also known as the straggler effect. This can impact scalability, especially in large distributed systems where node performance can vary significantly. In summary, the rule for updating the parameters, executed by the server at each iteration, can be described as follows [68]:

$$\begin{cases} \nabla L(\partial_t) = \frac{1}{N} \sum_{i=1}^{N} \nabla L_i(\partial_t) & \triangleright \text{ Gradient Aggregation} \\ \partial_{t+1} = \partial_t - \eta \nabla L(\partial_t) & \triangleright \text{ Parameter Update} \end{cases} \tag{2.8}$$

Where in the rule above, η denotes the learning rate, and N represents the number of worker nodes. The gradient $\nabla L_i(\partial_t)$ is computed by each worker node i based on its local data. The

aggregated gradient $\nabla L(\partial_t)$ is the average of the gradients from all workers, ensuring that the model parameters are updated consistently. This consistency ensures that all nodes work with the same model parameters in each iteration.

Asynchronous SGD (ASGD) ASGD, conversely, allows worker nodes to proceed with their computations and update model parameters independently of other nodes. This reduces idle times and can lead to better utilization of computational resources, thereby enhancing scalability. However, the lack of synchronization can result in inconsistencies, as updates may be based on stale gradients[6], potentially slowing down the convergence of the model or leading to convergence issues. In Summary, the update rule utilized in ASSD is[69]:

$$\partial_{t+1} = \partial_t - \eta \nabla L_i(\partial_t) \tag{2.9}$$

Unlike SSGD, in ASGD each worker node i independently computes the gradient $\nabla L_i(\partial_t)$ based on its local data and immediately applies this gradient to update the global model parameters ∂_t. This approach allows for faster updates but can introduce inconsistencies due to the varying staleness of gradients, as nodes do not wait for each other to synchronize their updates

The choice between Synchronous and Asynchronous SGD is crucial in designing effective DML systems. Synchronous SGD is typically preferred when consistency and simplicity are paramount, and the computational environment is relatively homogeneous, minimizing the straggler effect. On the other hand, Asynchronous SGD is advantageous in heterogeneous environments where reducing idle time and improving resource utilization is critical. The same applies to all the modified and improved variations stemming from these fundamental algorithms that comprise the cornerstone of DML algorithm implementations.

Concluding, as stated at the beginning of this subsection section 2.4.2 the architecture of a DML system can be decomposed into the three layers of *The Distribution Method (Parallelism)*, *the System's Communication Topology* and *the ML algorithm*. Therefore, researchers and practitioners of the field need always be aware of the characteristics, both merits and challenges, various designing options entail, in order to orchestrate efficient and useful DML Systems that can successfully satisfy any problem's requirements.

2.4.3 Challenges in Distributed Machine Learning

Distributed Machine Learning (DML) offers significant advantages in processing large datasets and complex models by leveraging parallelism and distributed resources. However, it also introduces several challenges, as depicted in table 2.3, that can impact efficiency, scalability, consistency, and security. These challenges are critical to understand as they set the stage for the transition to Federated Learning (FL), which addresses several of these issues.

Transition to Federated Learning Federated Learning (FL) builds on the principles of Distributed Machine Learning (DML) but addresses many of its inherent challenges, particularly those related to data privacy, communication overhead, and data heterogeneity. By preserving the training dataset locally, Federated Learning never utilizes the network for private data transmission, thereby negating any potential privacy infringements. Moreover, by enabling local training on edge devices and aggregating only the necessary updates at the end of a learning round, FL restricts network utilization and further enhances privacy.

Additionally, Federated Learning systems do not operate under the assumption of independent and identically distributed (i.i.d.) data, a common constraint in other DML system setups. Finally,

[6]The phenomenon where gradients are computed based on outdated parameters.

Challenge	Description	Addressed by FL
Communication Overhead	Frequent communication for gradient updates and synchronization can lead to significant overhead, particularly in environments with limited bandwidth or high latency.	Yes
Privacy and Security	Distributed systems are vulnerable to privacy breaches and security attacks, as data is transferred across nodes and potentially exposed to unauthorized access.	Yes
Data Heterogeneity	Data distributed across nodes may not be identically and independently distributed (non-IID), leading to biases and inconsistencies in model training.	Yes
Scalability	Managing communication and synchronization efficiently becomes challenging as the number of nodes increases, potentially incurring high costs.	Partially
Fault Tolerance	Ensuring reliability and fault tolerance in a distributed setting is complex, requiring robust mechanisms to handle node failures and network disruptions.	No
Straggler Effect	In synchronous training, faster nodes must wait for slower ones, leading to idle times and inefficient resource utilization.	No

Table 2.3. *Challenges of Distributed Machine Learning*

scalability is partially improved due to the overall reduction in communication overhead, as only model update transmissions are executed and no raw data transfers occur. This leads to more system resources being available. However, it should be noted that this performance improvement cannot be objectively measured in all cases and may sometimes be negligible.

This transition from DML to FL represents an evolution towards more efficient, scalable, and secure machine learning practices, leveraging distributed resources while mitigating the traditional challenges of DML systems. In summary, the challenges of DML portrayed at table 2.3 highlight the need for more advanced approaches like Federated Learning. FL addresses these issues by enabling decentralized data processing and focusing on privacy-preserving methods, making it a crucial advancement for modern machine learning applications, as will be further illustrated in the next section.

2.5 Federated Learning

In an era where cloud computing is the dominant paradigm for learning, Federated Learning (FL) has the potential to drive significant future changes in the industry. Despite the cloud computing market being dominated by tech giants like Google, Amazon, and Microsoft, Google was the first to recognize the emerging need for a democratized approach to machine learning. Taking a leading role in Federated Learning research, Google introduced the term in 2016 with a seminal paper on efficient Distributed Learning[70]. The significance of Google's approach cannot be overstated, as it eliminates the need for maintaining centralized data centers, allowing cloud resources to be allocated for various other purposes.

The term FL represents a significant shift in how machine learning models are trained and how data privacy is preserved. As an advanced paradigm within Distributed Machine Learning (DML),

FL enables training models across multiple decentralized devices or servers holding data samples locally, without the need to transfer this data to a central server. This approach addresses several critical challenges inherent in traditional centralized Machine Learning, including data privacy, communication overheads, and the handling of heterogeneous data and systems.

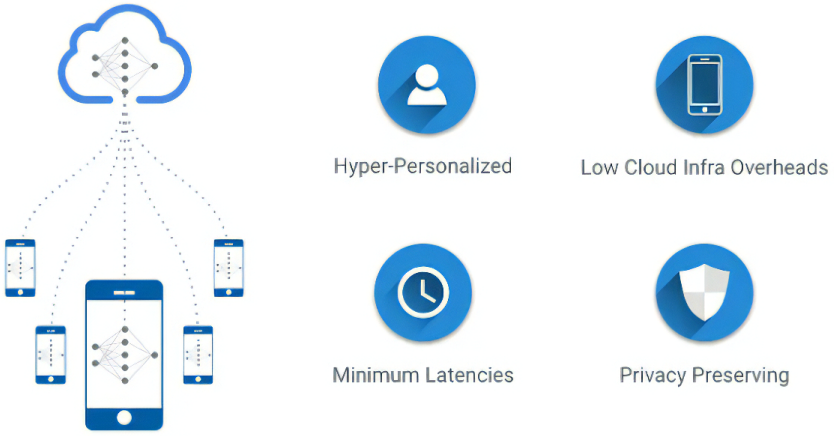


Figure 2.14. *Federated Learning Characteristics*

2.5.1 Principles of Federated Learning

Providing a solution to these issues FL emerged by enabling model training across decentralized devices or servers while keeping the data local. This method is particularly relevant in scenarios where data privacy is paramount, such as healthcare, finance, and personal devices like smartphones. A variety of FL applications can be located in everyday examples-namely image and video automatic categorization, personalized feeds of news or advertisements, and autocorrect and predictive text (f.i. Google's GBoard[11]).
The fundamental principles that determine FL's functionality are categorized as follows[71]:

- *Local Data Stays Local:* Data generated by devices or organizations remains on the local device, preventing the exposure of sensitive information to external threats or privacy breaches, as it does not need to be uploaded to a centralized server for processing.
- *Collaborative Model Training:* A central server distributes a global model to devices, which update it based on their local data. These updates, typically in the form of gradients or parameters, are sent back to the server and aggregated to enhance the global model iteratively, thus improving accuracy and utility without accessing the raw data.
- *Privacy by Design:* FL integrates privacy-preserving mechanisms such as secure multi-party computation and differential privacy, ensuring that model updates shared with the server do not disclose sensitive information about the underlying data or individuals. The privacy rules that are applied, are determined both by international organisms[7] and local authorities relevant to the region.
- *Efficiency and Scalability:* By processing data locally and exchanging only small model updates, FL reduces bandwidth requirements for model training, making it scalable across numerous devices and participants, each contributing to the refinement of the global model.

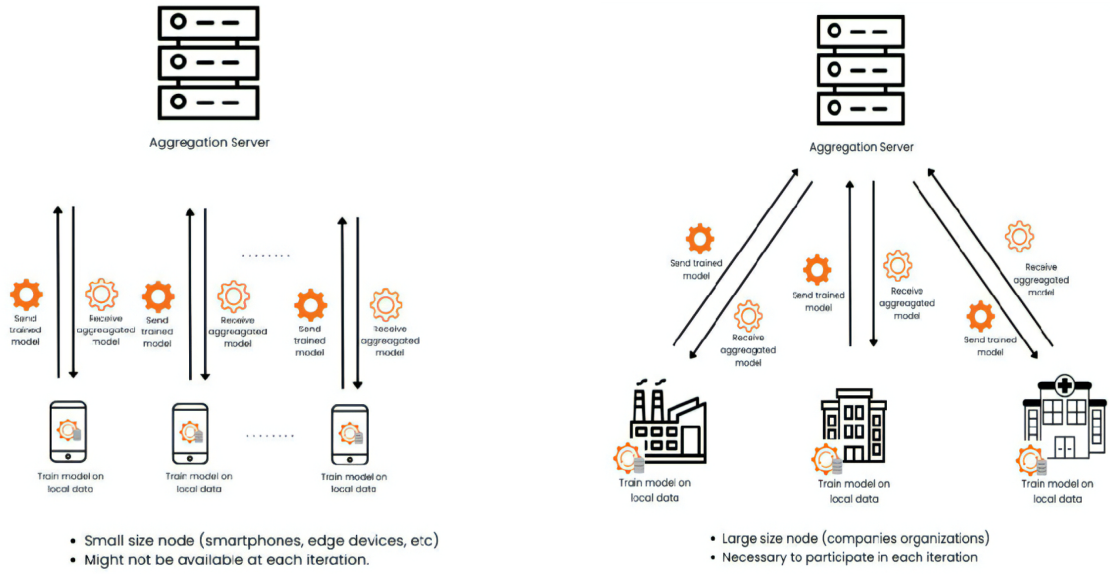
[7] Privacy preserving legislations such as the General Data Protection Regulation (GDPR) in the European Union and the California Consumer Privacy Act (CCPA) in the United States[5, 6].

2.5.2 Federated Learning Categorizations

Several criteria can be used to determine a holistic categorization of Federated Learning (FL) approaches[72]. For the purpose of this research study, the two principal factors are the scale of learning and the alignment of the distributed data. Based on the learning scale, the separation is executed into Cross-Device and Cross-Silo settings. Based on the data distribution, the categorization is between Horizontal and Vertical learning settings. The terms presented above will be explained next:

Learning Scale: The distinction is based on whether the federated learning system involves numerous personal or edge devices (cross-device) or a smaller number of institutions or organizations (cross-silo).

Alignment of the Distributed Data: The distinction is based on whether the datasets across entities share the same feature space but have different samples (horizontal) or share the same sample space but have different features (vertical).



(a) *Cross-Device learning* **(b)** *Cross-Silo learning*

Figure 2.15. *Federated Learning classification based on the learning scale*

Cross-Device Federated Learning Cross-device Federated Learning (FL) involves training models on a large number of edge devices such as smartphones and IoT devices (see fig. 2.15a). Each device holds a small, local dataset. This type of FL is particularly useful for applications where individual user data is sensitive and thus cannot be shared directly. The primary challenges include managing the diversity in device capabilities, ensuring efficient communication, and maintaining data privacy. Additionally, a device may not be available for training in all fitting rounds, which adds complexity to the training process. Techniques like asynchronous updates and efficient gradient compression are often employed to address these challenges[15, 70].

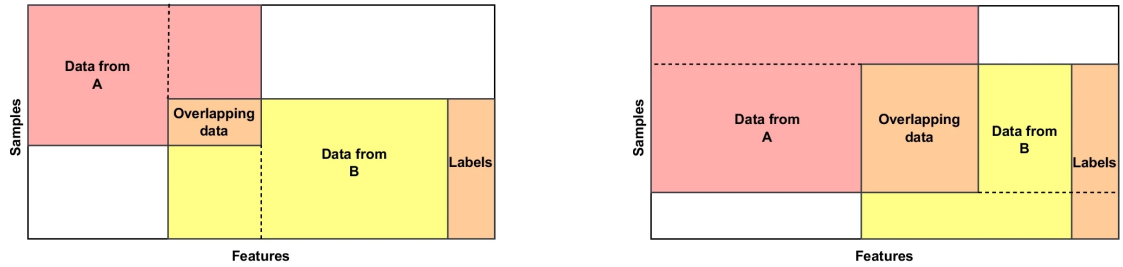
Cross-Silo Federated Learning Cross-silo FL is used when a small number of organizations, such as hospitals or financial institutions, collaborate to train a machine learning model (see fig. 2.15b). Each organization (silo) has a larger dataset, but data cannot be shared due to privacy

concerns. This method often aligns with regulatory requirements such as GDPR and CCPA. Challenges include managing non-IID data distributions across silos and ensuring secure aggregation of model updates. Techniques like secure multi-party computation (SMPC) and differential privacy are crucial in this context[13, 73].

Horizontal and Vertical Federated Learning

Horizontal Federated Learning (HFL) HFL, or sample-based FL, occurs when different entities (devices or institutions) have datasets with the same feature space but different samples. For instance, multiple hospitals might have patient records with the same attributes. HFL is commonly used in both cross-device and cross-silo settings. The main challenge is handling non-IID data, where each client's data distribution may vary significantly. Approaches like data augmentation and federated averaging (FedAvg) are often used to mitigate these issues[3].

Vertical Federated Learning (VFL) VFL, or feature-based FL, occurs when entities have datasets with different feature spaces but the same sample space. For example, one organization might have transaction data, while another has demographic data about the same customers. VFL is typically used in cross-silo settings. The primary challenge is ensuring secure and efficient computation when combining different types of data. Secure computation techniques and feature alignment methods are critical to address these challenges [13, 15].



(a) *Horizontal Federated Learning* **(b)** *Vertical Federated Learning*

Figure 2.16. *Federated Learning classification based on alignment of distributed data[72]*

Based on the aforementioned criteria, the system explored in this research is categorized as Cross-Device Horizontal Federated Learning (HFL).

2.5.3 Federated Learning Methodology

Architecture

In a Federated Learning setup, the architecture can be decomposed into four principal components: client devices, the central server, communication protocols, and security mechanisms. An overview of each component's functionality will be presented for a typical Federated Learning architecture, based on the cumulative analysis provided by multiple literature sources [3, 13, 70, 74]. Detailed explanations of certain terms will be avoided if they are not directly relevant to the research objectives. In such cases, only a conceptual illustration will be provided.

Client Devices: Client devices are the edge devices or local servers that generate and hold the local data. These can include smartphones, IoT devices, or organizational databases. Each client device performs local training on its dataset, updating the model's parameters based on local data. This helps preserve data privacy since the raw data never leaves the device. After training locally,

the clients send their model updates (e.g., gradients or parameters) to the central server. These updates are computed using the local data, ensuring that sensitive or personal information is not exposed. The heterogeneity of these devices, in terms of computational power and network connectivity, presents a challenge that must be managed by the system.

Central Server: The central server acts as a coordinator that aggregates model updates from multiple client devices and updates the global model. Essentially, the server collects model updates sent by the clients, aggregates these updates to refine the global model, and then redistributes the updated global model back to the clients for further training iterations. This process ensures that the global model improves iteratively based on the decentralized data from all participating clients. The central server must handle the challenges of aggregating updates efficiently and securely to maintain the model's performance and integrity.

Communication Protocols: The communication protocol defines the rules and methods for data exchange between client devices and the central server. Efficient communication protocols are essential to minimize latency and bandwidth usage during the model update exchanges. Protocols in FL often incorporate techniques to reduce the volume of data transferred, such as model compression and gradient sparsification. Asynchronous communication methods can handle the varying availability and performance of client devices, reducing overall training time and mitigating the straggler effect.

Security Mechanisms: Security mechanisms are critical in FL to ensure that the data and model updates remain confidential and tamper-proof. These mechanisms protect against potential attacks that could compromise the privacy and integrity of the data being processed. Common security measures include encryption, differential privacy, and secure multi-party computation (SMPC). Encryption ensures that data transmitted between clients and the server is protected from eavesdropping. Differential privacy adds noise to the model updates to prevent the reconstruction of sensitive data. SMPC allows multiple parties to compute a function over their inputs while keeping those inputs private, further enhancing the security of the federated learning process.

2.5.4 Problem Formulation in Federated Learning

This section provides a clear overview of the FL problem statement based on the Almanifi et al. article (2023)[75]. In the broader field of Machine Learning (ML), the primary problem is to learn a model that maps a set of input data x to a set of desired outputs y based on a number of training examples. The performance of this model is evaluated using a loss function $f_i(w)$, which measures the error between the model's predictions and the actual desired outputs. For a set of n input-output pairs $\{x_i, y_i\}_{i=1}^{n}$, the goal is to find the optimal weight vector(s) w that minimizes the loss function. An example of this in a linear regression problem is [8]:

$$f_i(w) = \frac{1}{2}(x_i^T w - y_i)^2, \quad y_i \in \mathbb{R} \text{ and } x_i \in \mathbb{R}^d \tag{2.10}$$

Although loss functions vary across different ML techniques, the problem generally can be formulated as an optimization problem for a finite-sum of functions, expressed as:

$$\min_{w \in \mathbb{R}^d} f(w) \quad \text{where} \quad f(w) = \frac{1}{n} \sum_{i=1}^{n} f_i(w) \tag{2.11}$$

[8] Linear regression typically employs Mean Squared Error (MSE) as its loss function.

In Federated Learning, the training process minimizes the loss function locally through gradient descent before the weights (w) are communicated to the central server. Consider an FL system with a fixed number of clients K, where each client hosts a partition of the data n_k, indexed by $\mathcal{P}_k = \{1, 2, \ldots, K\}$. Upon receiving all local parameters, the global server aggregates w_i, i representig the client index, to update the new global model. The aggregation mechanism sets the global objective. For example, in FedAVG, one of the most commonly used aggregation methods in FL, the loss function is updated to:

$$f(w) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(w) \quad \text{where} \quad F_k(w) = \frac{1}{n_k} \sum_{i \in \mathcal{P}_k} f_i(w) \tag{2.12}$$

Thus, the problem in FL becomes learning the aforementioned model in a distributed manner without exchanging raw data.

Federated Learning Process

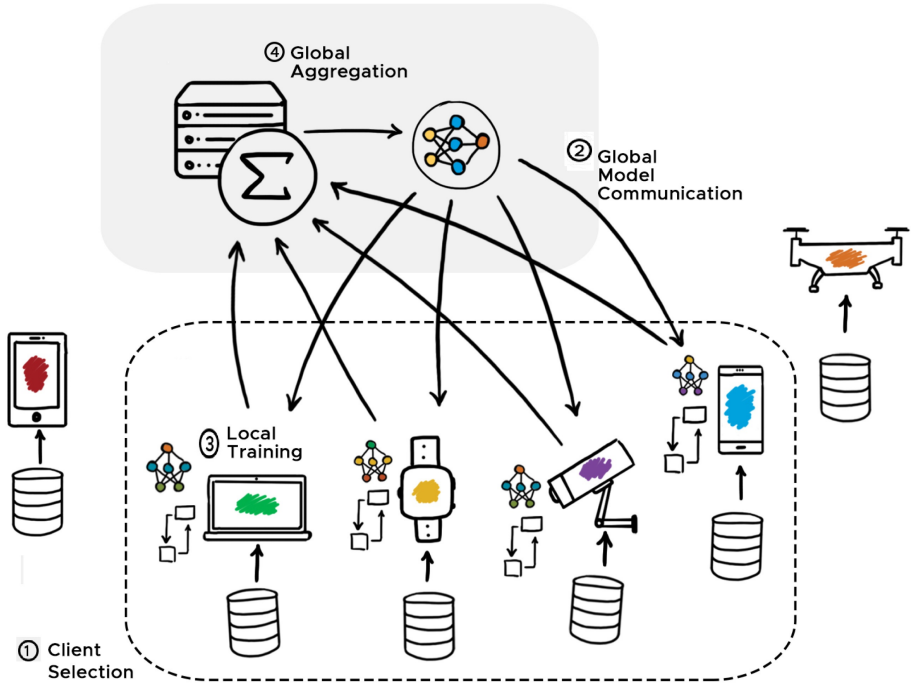


Figure 2.17. *General Federated Learning training process[75]*

In distributed machine learning setups, such as Federated Learning, processes are often viewed as continuous, meaning that the system continuously updates and refines the model parameters based on incoming data. This continuous process allows for dynamic adjustments and real-time improvements, making the system more adaptive and responsive to variations in the data[12, 75]. Provided that, in the iterative training process, each iteration/round can be separated into four major steps as displayed in fig. 2.17:

1. *Client Selection:* Selection strategies vary, as client selection can be executed either randomly, or with prioritization criteria in mind. Depending the FL setup, client selection can be conducted at two different stages: either before sending the global model to the clients (first step) or after local training (third step). *First Step Selection*: The central manager selects a subset of clients before sending the global model. This pre-selection can be used to restrict the number of clients that participate in training, while also providing the opportunity to filter clients based

on their updates quality. *Third Step Selection*: After local training, the server selects a subset of clients to transmit their model updates. This step prevents diminishing returns seen when the full set of clients is used. In our implementation client selection is conducted before the global model communication.

2. *Global Model Communication:* The process begins with available clients notifying the server of their readiness to participate. The server then communicates the global model to all participating clients, signaling the beginning of the communication round. This step involves broadcasting the global model's parameters, which include weights and biases, along with any relevant metadata such as hyperparameters (e.g., learning rate, batch size).

3. *Local Training:* Each client trains the model locally using its own dataset. During this step, the clients adjust the global model parameters based on their local data, resulting in a *Model Update*. The training process is conducted according to the instructions and hyperparameters provided by the central server. The objective function of the training process is denoted as f. For a selected client n from the set of clients N, the client aims to minimize its loss function L, which depends on the global model weights w_g and the local data d_i:

$$f_n(w_g) = \arg\min_{n \in N} L(w_g; d_n) \tag{2.13}$$

   After training, clients need to transmit their model updates back to the central server for aggregation. The model updates may undergo additional processing to enhance efficiency and security, as outlined in section 2.5.3. Finally, the processed model updates are transmitted to the server, and the clients remain inactive until selected again.

4. *Global Aggregation:* Upon receiving the updated parameters from each selected client, the server aggregates them into a global model using a chosen aggregation mechanism. Various strategies can be employed, such as the FedAVG method, which averages the updates from the clients. The global objective is to minimize the overall loss function as shown in eq. (2.11). Once the model is updated, the communication round ends. The server then evaluates the global model's performance using metrics over a public test set. Based on this evaluation, training may be terminated if the desired accuracy is achieved or the preset number of rounds is completed; otherwise, a new communication round begins.

The above functionality describes a synchronous Federated Learning setup, which aligns with the thesis implementation. However, as highlighted earlier in order to improve a distributed system's performance, asynchronous methods can be applied too, mitigating the straggler effect.

While programming an asynchronous learning system is taxing and requires careful considerations about out-of-sync updates, the fundamental logic remains the same with modifications primarily pertaining to the timing and coordination of these steps. To be exact, In an asynchronous Federated Learning setup, client selection remains dynamic, but clients are allowed to start and finish their local training at different times. This flexibility means that the server can receive and aggregate updates from clients as they become available, without waiting for all selected clients to complete their training. Consequently, global model communication is also more fluid, with the server sending the global model to clients on an as-needed basis, rather than all at once. Local training on each client still follows the same process, but clients send their updates to the server independently upon completion. The server continuously aggregates these updates using the chosen aggregation mechanism, such as weighted averaging and updates the global model iteratively.

2.6 Networking in Federated Systems

2.6.1 Network Topologies

The underlying network infrastructure plays a crucial role in the performance and scalability of Federated Learning FL and Distributed Machine Learning systems, in general. Three of the most prevalent topologies employed in DML Systems were described earlier in section 2.4.2. Likewise, it can be stated that by being a subset of DML systems Federated Systems are constructed using the same fundamental topologies. That said, FL Systems are indeed typically arranged in star, hierarchical tree, ring, or peer-to-peer topologies.

Recapitulating briefly in regards with FL, *star topologies* are a simplification of parameter server setups, where N clients are directly connected with one central server (parameter server) that coordinates the learning procedure. The star topology is advantageous for its simplicity and ease of implementation. However, a singular server can become a bottleneck and a single point of failure, limiting the system's scalability and robustness.

Hierarchical tree topologies, differ from star topologies due to the fact that clients do not always communicate directly with the server. Instead, they might be organized in smaller clusters of M clients where $M < N$, that transmit their model updates to intermediate stations responsible for aggregating local model updates, thus communicating the intermediate aggregation results to the server. This hierarchical approach can improve scalability by distributing the aggregation workload and reducing the communication burden on the central server. Additionally, it can enhance fault tolerance by localizing the impact of network failures.

Ring setups indicate that each node communicates directly with its two neighbors. This way, communication is conducted in a circular manner, presenting a peer-to-peer approximation approach where the coordination responsibility is distributed to the workers. This topology can provide robustness and fault tolerance, as the failure of a single node does not disrupt the entire network. However, it may suffer from higher latency as the number of nodes increases due to the sequential nature of communication.

Finally, In a *fully peer-to-peer* setup, clients communicate directly with each other broadcasting their model updates without a central coordinating authority. Protocols such as gossip algorithms are used to propagate updates throughout the network. This approach can enhance privacy and decentralization but may require more complex coordination mechanisms.

Overall, it becomes obvious that the selection of each of these topologies for the design of an FL system entails certain merits and challenges concerning the communication efficiency.

2.6.2 Communication in Federated Systems

Communication Protocols

The communication protocol defines the rules and methods for data exchange between client devices and the central server. Efficient communication protocols are essential to maximize performance during model exchange procedures. There is a plethora of contemporary technologies with diverse characteristics being utilized in Federated communication, with most primary examples being gRPC, Protocol Buffers (Protobuf), WebSockets, and HTTP/2 and TCP Sockets.

- *gRPC* is a high-performance, open-source Remote Procedure Call (RPC) framework that can run in any environment. It uses HTTP/2 for transport, Protocol Buffers (protobuf) as the interface description language, and provides features such as authentication, load balancing, and more. The efficient serialization and transport mechanisms of gRPC make it well-suited for federated learning scenarios where low latency and high throughput are crucial.

- *HTTP/2* is a major revision of the HTTP network protocol, focusing on performance improvements. It introduces features like multiplexing, header compression, and server push. This protocol has become an industry standard due to its unmatched efficiency.

- *Protobufs (Protocol Buffers)* is a language-neutral, platform-neutral, extensible mechanism for serializing structured data. Protobuf is used to define the structure of messages-namely model parameters, training configurations, and results exchanged between the server and clients. This ensures efficient and compact communication.

- *WebSockets* provide a full-duplex communication channel over a single, long-lived TCP connection. They are useful for real-time applications requiring continuous data exchange. That makes WebSokcets the first choice when maintaining a persistent connection between the server and clients is beneficial. This can be particularly useful in real-time monitoring and control of the FL process.

- *TCP Sockets* offer a reliable, connection-oriented communication protocol that ensures data integrity and delivery. TCP handles packet loss, data corruption, and data reordering, making it suitable for scenarios where reliability is paramount. In federated learning, TCP sockets can be used to establish robust connections between the server and clients, ensuring that critical updates are transmitted accurately and efficiently.

Connectivity Settings

The performance of federated learning systems is significantly influenced by the type of connectivity used. Different network types offer distinct advantages and are preferred in various scenarios:

- *Ethernet*: Provides high reliability and consistent bandwidth, making it ideal for environments requiring stable, high-speed communication, such as data centers and research facilities where large-scale model training occurs.

- *Wi-Fi*: Offers flexibility and mobility, suitable for residential or office settings where devices are not fixed and may move around. It is preferred in scenarios like smart homes or collaborative workspaces where user devices frequently change locations.

- *Point-to-Point Connections*: Provide direct and reliable communication paths, preferred in specialized settings such as direct communication between servers in a data center or between two devices in close proximity that require secure and fast data exchange without intermediate routers.

- *Cellular Networks (4G/5G)*: Useful for mobile and remote applications where devices are dispersed over large geographical areas. This is ideal for applications like connected vehicles or remote health monitoring systems, where data needs to be collected from widely distributed sources.

Regardless of the selected communication protocol, and connectivity setting, communication overheads can always be reduced by minimizing the size of transmitted data. Techniques such as quantization, compression and efficient encoding can significantly decrease the amount of data exchanged between clients and the server, thus reducing latency and improving overall system efficiency.

2.6.3 Communication Performance Metrics

In FL systems, evaluating communication performance is paramount to understanding the efficiency and effectiveness of distributed learning processes. Several critical metrics are used, to assess communication performance, each contributing to a comprehensive analysis of the system's capabilities and limitations.

Latency (*L*) is a fundamental metric, representing the time interval required for a message to travel from the source node to the destination node. It is indicative of the system's responsiveness. Mathematically, latency is defined as:

$$L = t_r - t_s$$

where t_s is the time a message is sent, and t_r is the time it is received.

Bandwidth (*B*) measures the maximum rate at which data can be transmitted over a network connection within a specific time frame, reflecting the network's capacity to handle data transfers. It is given by:

$$B = \frac{D}{T}$$

where D is the total amount of data that can be transferred, and T is the theoretical time taken for the transfer.

Throughput (*T*) is closely related to bandwidth but focuses on the actual rate of successful data transfer over the network, considering factors such as network congestion and protocol overhead. In other words, the first one calculates the maximum achievable transfer speed of the channel, while the second represents the actual calculated value attained in an experiment. It is defined as:

$$T = \frac{D_r}{T_r}$$

where D_r is the actual amount of data transferred, and T is the actual time taken for the transfer.

Packet Loss Rate (PLR) is the percentage of packets lost during transmission, which can significantly impact the performance of FL systems. It is calculated as:

$$PLR = \frac{P_{sent} - P_{received}}{P_{sent}} \times 100\%$$

where P_{sent} is the number of packets sent, and $P_{received}$ is the number of packets received successfully.

Dropout Rate (*DR*) is a significant metric in federated learning, indicating the percentage of clients that fail to complete the training process due to various reasons such as connectivity issues or resource constraints. It is defined as:

$$DR = \frac{N_{dropout}}{N_{total}} \times 100\%$$

where $N_{dropout}$ is the number of clients that dropped out, and N_{total} is the total number of clients.

The Computation-Communication Ratio (*CCR*) assesses the balance between computational load and communication overhead in federated systems. It is crucial for optimizing performance and resource allocation. *CCR* is defined as:

$$CCR = \frac{C_{comp}}{C_{comp} + C_{comm}}$$

where C_{comp} is the time spent on computation, and C_{comm} is the time spent on communication.

Effective evaluation of these metrics helps in identifying bottlenecks and optimizing the perfor-

mance of FL systems. By understanding the impact of latency, bandwidth, packet loss, dropout rates, and the computation-communication ratio, researchers and practitioners can enhance the efficiency and robustness of distributed learning processes. These metrics provide insights into how well the system can scale and adapt to varying network conditions and client capabilities.

2.6.4 Simulating Real-World Networks

Network simulators can play a pivotal role in Federated Learning (FL) research by enabling researchers to test and evaluate the performance of FL systems under various network conditions. This capability is significant because setting up FL environments with numerous devices in a controlled laboratory setting is challenging. Simulators such as *NS-3*, *OMNeT++*, and *Mininet* can emulate real-world network environments, offering valuable insights into how different network characteristics influence the training process.

Mimicking Real-World Network Conditions

Simulators can accurately replicate several aspects of real-world networks, including:

- *Network Congestion:* By simulating varying levels of network traffic, researchers can observe how congestion impacts data transmission and model updates in FL systems.
- *Latency Variability:* Simulators can introduce different latency values to examine how delays in communication affect the synchronization and convergence of models.
- *Bandwidth Fluctuations:* Emulating changes in available bandwidth helps in understanding the effect on the speed and efficiency of data exchanges between clients and the central server.
- *Error Rates:* Introducing different error rates across multiple clients can mimic real-life scenarios where devices have varying communication capabilities and reliability.
- *Network Topologies:* Simulators can model different network topologies, such as star, ring, and hierarchical structures, to study their impact on the performance of FL systems.

Measuring Contributions from Simulations

Through the use of network simulators, researchers can measure several key performance metrics that provide insights into the effectiveness and robustness of FL systems:

- *Impact on Model Convergence:* By analyzing how network conditions such as latency and bandwidth influence the speed and accuracy of model convergence, researchers can identify bottlenecks and optimize communication strategies.
- *System Robustness:* Simulating scenarios with multiple clients experiencing varying error rates allows for the assessment of the system's robustness and its ability to handle client dropouts or unreliable connections.
- *Scalability Analysis:* Researchers can observe how the FL system scales with the number of clients, particularly when clients have different communication capabilities. This helps in understanding the limits and potential optimizations for large-scale deployments.
- *Topology Influence:* Evaluating the effect of different network topologies on system performance provides insights into the most efficient structures for specific FL applications.

In summary, network simulators offer significant capabilities for Federated Learning (FL) research, allowing for detailed studies of various network conditions and their impact on the training process. These tools help identify potential issues and optimize FL systems, ensuring efficient, scalable, and robust performance in real-world deployments.

2.7 Summary & Transition to Related Work

The theoretical background overview provided in this chapter laid the groundwork for highlighting the principal elements, definitions and contempory challenges relevant to our *Federated Learning* research. We began by exploring the broader concept of *Machine Learning* exploring the fundamental principles and objectives of the training process, proceeding with *Artificial Neural Networks* and deep learning to establish a basis for understanding complex model architectures and contemporary learning paradigms. Then our analysis took a turn towards *Optimization Techniques*, crucial for enhancing the learning efficiency of sophisticated, multi-parameter model architectures like the ones employed in *Federated Learning* setups.

Subsequently, the discussion focused on *Distributed Learning Systems*, highlighting their significance in managing learning with immense datasets distributed in a multitude of worker nodes, while preserving computation efficiency. This naturally led to an in-depth analysis of *Federated Learning*, which builds on distributed learning principles to enhance data privacy and minimize communication overhead by allowing decentralized data processing and model updates. To conclude, a brief demonstration of *Networking in FL* was provided, studying the impact of different topologies and characteristics in the learning process.

All things considered, this in-depth survey of the existing literature provides the necessary terminology and background for demonstrating the proposed ideas outlined in the *Introduction*. The following chapter presents the preliminary work, which forms the fundamental basis for the development and evaluation of a Federated Learning procedure. This includes employing an optimized synchronization technique to further mitigate communication overheads, offering a unified approach for managing the learning process in a user-friendly framework for federation, and simulating the impact of various network parameters on performance. Topics discussed in the next chapter include: *datasets used in federation*, *common CNN architectures*, *algorithmic implementations of FL*, and relevant *synchronization optimizations*. The chapter concludes with an *FL simulator* tha studies the synergy between a framework for data management in Federated setups and a network simulator for enforcing realistic communication conditions.

Chapter 3

Related Work

3.1 Datasets in Federated Learning Research

Datasets comprise a fundamental factor in Machine Learning, including fields such as Federated Learning. Essentially, they offer training samples along with their corresponding labels, providing the algorithm with the necessary input to be able to extract underlying model patterns, relationships, and representations essential for accurate and efficient generalization. A good dataset for machine learning needs to satisfy the following requirements:

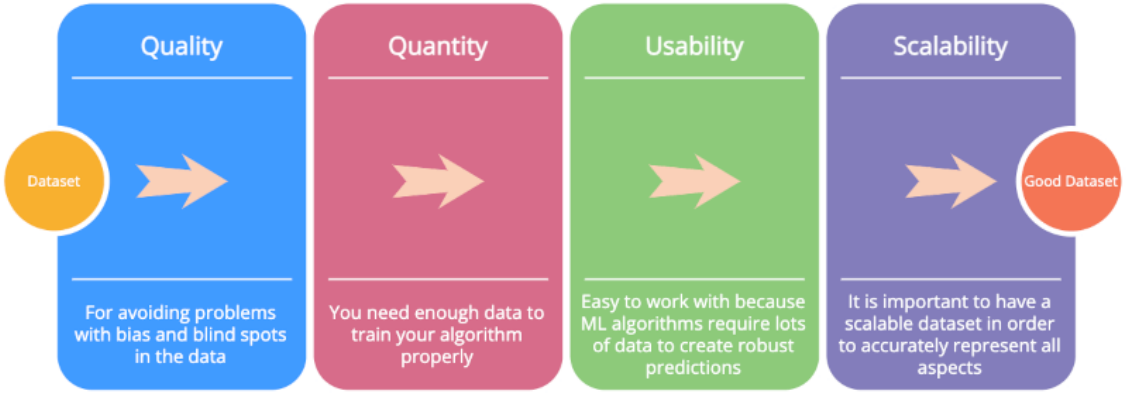


Figure 3.1. *Features of a good dataset for Machine Learning*

Quality samples in datasets are accurate, reliable, and contain a balanced distribution among the data classes, ensuring that the model learns from trustworthy and informative examples. Quantity is necessary with ML algorithms requiring a vast number of samples to achieve generalization, offering diverse coverage of the underlying data distributions to avert overfitting and achieve convergence. Usability refers to the dataset being properly formatted and accompanied by clear metadata facilitating seamless utilization by researchers and practitioners. Finally, scalability is required to be able to accommodate varying volumes of data without compromising performance or efficiency.

3.1.1 Benchmark Datasets

The process of creating a proper dataset for machine learning tasks, especially in the context of federated learning, is non-trivial. It involves meticulous curation, preprocessing, and validation to ensure that the dataset accurately represents the underlying data distribution and facilitates meaningful model training. Given the complexity and challenges associated with dataset creation, several benchmark datasets are utilized in the FL community to evaluate the performance of

federated learning algorithms. Some examples relevant to our research's implementation are presented:

- **MNIST**: MNIST is a dataset of handwritten digits widely used for image classification tasks. It consists of 28×28 grayscale images of digits from 0 to 9, with corresponding labels.
- **Fashion-MNIST**: Fashion-MNIST is a dataset of clothing articles used for image classification tasks, similar to MNIST but with more diverse classes. It consists of 28×28 grayscale images of fashion items such as shirts, shoes, and trousers.
- **CIFAR-10**: CIFAR-10 is a dataset of small images across ten classes, often used for image recognition tasks. It contains 60,000 32×32 color images, with 6,000 images per class.
- **CIFAR-100**: CIFAR-100 is an extension of CIFAR-10, containing 100 classes of objects. It provides a more challenging classification task compared to CIFAR-10, with finer-grained categories.

These benchmark datasets serve as standardized tasks for evaluating federated learning algorithms, allowing researchers to assess model performance across different domains. By utilizing well-established datasets like MNIST, Fashion-MNIST, CIFAR-10 and CIFAR-100 researchers can compare the effectiveness of their FL implementations, measure their generalization capabilities, and identify areas for improvement in federated learning methodologies. For those reasons, such datasets were preferred in our study.

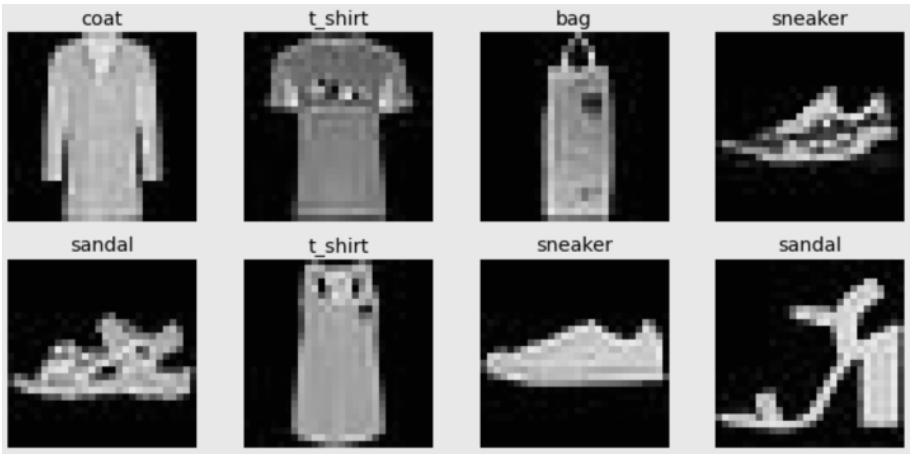


Figure 3.2. *Illustration of Fashion-MNIST samples*

3.1.2 Real-world Datasets

While benchmark datasets offer controlled environments for evaluating federated learning algorithms, real-world datasets present unique challenges and opportunities, essential for real-life Federated System implementations. Benchmark datasets provide some intuition about system performance but do not encompass fundamental issues characteristic of realistic scenarios, such as heterogeneous distribution, data privacy regulations, and continuous data generation.

As Federated Learning expands into real-life applications in *healthcare*, *mobile apps*, and *finance*, the need for appropriate datasets for these applications increases. Examples include:

- **Healthcare Data**: Patient records, medical images, and sensor data collected from various healthcare institutions.

- **IoT Data**: Sensor readings, device logs, and telemetry data generated by Internet-of-Things (IoT) devices.
- **Financial Data**: Transaction records, credit scores, and market data from banking and financial institutions.

3.1.3 IID VS. non-IID Data Distributions

Two crucial categories of data distribution can be identified in Machine Learning settings.

Independent and Identically Distributed (IID) In an IID data distribution, each client's data is drawn from the same underlying distribution, and the data samples are independent of each other. That is to say, each sample is drawn independently from the same probability distribution and exhibits identical statistical properties with the rest, representing the overall population distribution.

Non-IID Non-IID data distributions occur when each client's data sample follows a different distribution, either due to variations in data sources, data collection methods, or client-specific characteristics.

In much of the traditional *Distributed Machine Learning Systems* research, IID (Independent and Identically Distributed) data distributions are frequently assumed. In such scenarios, the central server decentralizes data so that each client possesses a representative subset of the overall dataset. However, this assumption often overlooks the inherent complexities of real-world data distributions, particularly in federated systems. In federated learning, where data remains decentralized across multiple clients, achieving true IID data distributions is often impractical or unrealistic.

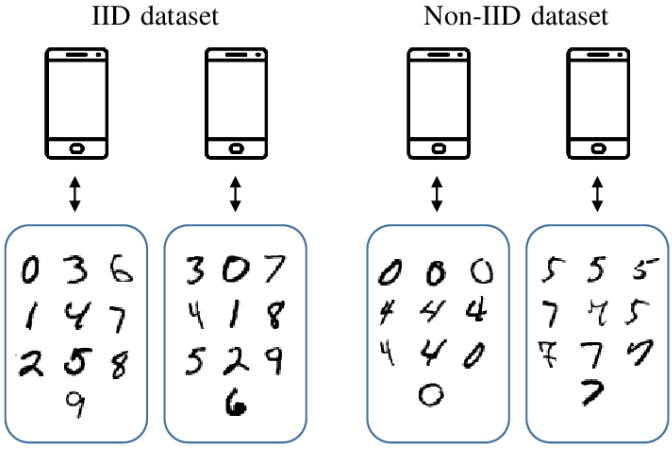


Figure 3.3. *Example of IID and non-IID distributions of MNIST: In the IID dataset, each device contains a similar and random sample of digits* (0 – 9)*, ensuring that the data distribution on each device is similar to the overall distribution. In the Non-IID dataset, each device contains digits that are not uniformly distributed; instead, each device has a concentration of certain digits. For example, one device has mostly 0s, another mostly 7s, and so on*

In Federated Learning, data distributions across clients can significantly impact the training process and model performance, thus, both cases should be studied. IID distributions provide a simplified benchmark for evaluating federated learning algorithms, while non-I.I.D reflect the inherent heterogeneity in practical deployments. The conjunction of the two ultimately ensures

that Federated Learning algorithms are not only theoretically sound but also practical and effective in real-world settings.

3.2 CNN Architectures

The never-ending advancement of ANN architectures has resulted in remarkable progress in image classification, utilizing innovative, and advanced CNNs. Over the years, various groundbreaking models have been introduced, each implementing novel techniques to enhance deep learning performance. The above statement can be additionally verified by the outstanding results in regards to the top-5 error rate accomplished in competitions such as ImageNet's Large Scale Visual Recognition Challenge (ILSVRC) where the error rate plummeted to less than 7% even from 2014[76]. In this section, the aim is to present the primary examples that have paved the way for these improvements from the inspiration startpoint of *LeNet-5* to 2014 and 2015 ILSVRC competition standouts in *VGG16* and *ResNet* respectively.

3.2.1 LeNet-5

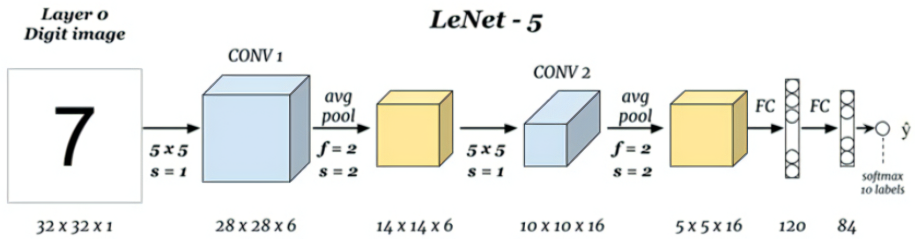


Figure 3.4. *LeNet-5 architecture*

LeNet-5, developed by Yann LeCun and his colleagues in the late 1990s[77], is widely regarded as the pioneer of modern Convolutional Neural Networks. Designed primarily for handwritten digit recognition, specifically the MNIST dataset, LeNet-5 laid the foundation for subsequent advancements in deep learning. The architecture consists of two sets of convolutional and subsampling layers, followed by three fully connected layers. Observing fig. 3.4, the original LeNet architecture utilized average pooling, which was a standard of the era, yet with current knowledge max pooling layers would be preferred, producing slightly improved results. However, for this analysis the vanilla version, as introduced in the original paper, is showcased.

The simple yet effective design of *LeNet-5* demonstrated the potential of CNNs in automatically learning hierarchical features from raw pixel data, a concept that has been fundamental to later architectures. LeNet's success in recognizing handwritten digits was revolutionary for its era and is still regarded as one of the most important historically CNN architectures.

3.2.2 VGG16

VGG16, introduced by the Visual Geometry Group and specifically A. Zisserman and K. Simonyan from the University of Oxford[78], based its implementation on the notion that the performance of a deep neural network can be improved by increasing the model's depth. To be precise, the number 16 on the ann's names was assigned for the 16 parameter layers that comprise the model, 13 of which are convolutional layers and the rest 3 are fully connected layers. VGG16 is renowned for its simplicity and robust performance in a variety of computer vision tasks. Its archi-

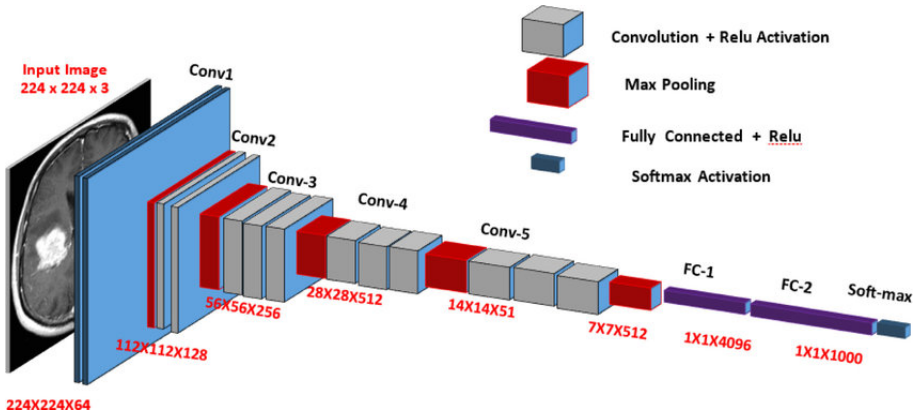


Figure 3.5. *VGG16 architecture*

tecture incorporates stacked convolution layers followed by max-pooling layers, with progressively increasing depth.

The primary novelty of the model's architecture is that it mitigates the network's parameters by leveraging 3×3 kernels with a stride of 1 to replicate a kernel of any size. Essentially, the dimensions of the output of any convolved layer can be achieved by stacking several smaller layers.

Consider an example to understand this logic. A 5×5 image is convolved with a 5×5 kernel, producing an output matrix of 1×1. If the same image is convolved by a 3×3 kernel, it produces a 3×3 matrix, which can be further convolved by another 3×3 kernel. The result will be an output matrix of 1×1. In this example, both processes result in a 1×1 matrix, yet the single kernel requires $5 \cdot 5 = 25$ parameters, while the two stacked convolution layers require $3 \cdot 3 + 3 \cdot 3 = 18$ parameters.

Following the innovative example of AlexNet-the 2012 ILSVRC winner- VGG16 employed ReLu activation functions in all the layers, reducing training time, while leveraging SoftMax function in the output layer to normalize the probability of the classes.

Combining the aforementioned elements, this architecture managed to classify with great precision images to 1000 object categories and became a significant paradigm in the field of ConvNets, still being utilized in systems that conduct deep learning.

3.2.3 ResNet

Residual Networks (ResNet), introduced by Kaiming He et al. in 2015[79], addressed the degradation problem in very deep networks. ResNet's key innovation is the introduction of residual learning blocks, or shortcuts, which add the input of the block directly to its output, allowing the network to learn identity mappings. These connections help prevent the vanishing gradient phenomenon during backpropagation and mitigate accuracy saturation, enabling the effective training of much deeper networks compared to previous architectures.

ResNet variants, such as ResNet-50, ResNet-101, and ResNet-152, have achieved remarkable success, with ResNet-152 achieving a top-5 error rate of 3.57% in the ILSVRC 2015, setting new performance standards. The success of ResNet demonstrated that increasing network depth could lead to better performance, provided that appropriate architectural adjustments are made to ensure effective training.

The architectures of LeNet-50, VGG, and ResNet represent significant milestones in the evolution of CNNs, each contributing foundational ideas and techniques that have propelled the field forward.

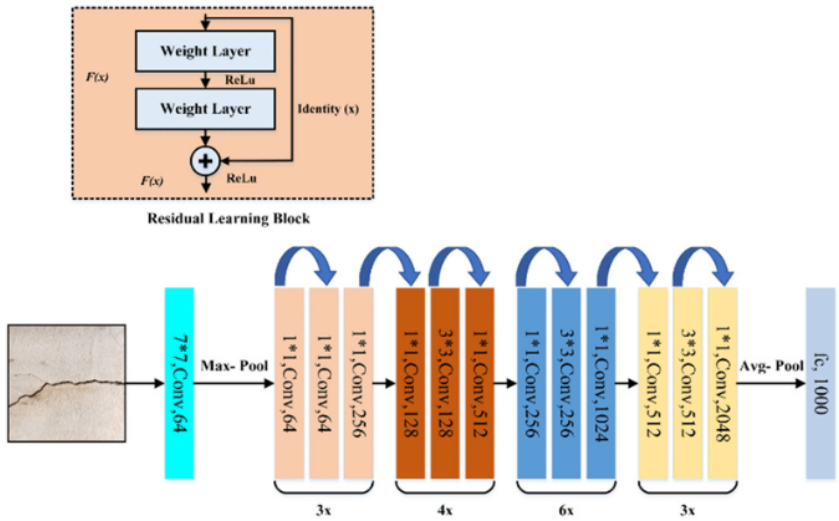


Figure 3.6. *ResNet50 architecture with residual learning block illustration*

3.3 Federated Learning Algorithms

In this section, we delve into the foundational algorithmic implementations that have shaped Federated Learning research, providing a structural background for subsequent efforts. We begin by discussing *FedSGD*, the simplest Federated Learning algorithm, which set the stage for more advanced methods. Following this, we introduce *FedAvg*, an algorithm that introduced several key innovations and serves as the basis for more sophisticated and optimized approaches currently employed in the domain. By concluding the analysis of these two principal algorithms, we will highlight the shortcomings of contemporary Federated Learning implementations.

Before focusing on the algorithmic implementations a common notation ground is laid with table 3.1:

Notation	Definition
w_t	Model weights on communication round # t
w_t^k	Model weights on communication round # t on client k
K	Number of all clients
C	Fraction of clients performing computations in each round
E	Number of training passes each client makes over its local dataset on each round
B	The local minibatch size used for the client updates
η	The learning rate
$\mathcal{P}_k$	Set of data points on client k
n_k	Number of data points on client k
$f_i(w)$	Loss $l(x_i, y_i; w)$ i.e., loss on example (x_i, y_i) with model parameters w

Table 3.1. *Notation used in Federated Learning algorithms*

3.3.1 Federated Stochastic Gradient Descent (FedSGD)

Stochastic Gradient Descent (SGD), has shown great results in deep learning, making it a foundational algorithm for many optimization problems. Given its effectiveness, researchers decided to base the Federated Learning training algorithm on SGD as a baseline. In traditional Stochastic Gradient Descent, gradients are computed on a random subset of the total dataset and then used

to update the model parameters. This process is repeated iteratively, making it computationally efficient for centralized learning.

However, applying SGD naively to the federated optimization problem involves a single batch gradient calculation on a randomly selected client per round of communication. This way each selected client participates in training using the local data points. While this approach is computationally efficient, it requires a large number of communication rounds to achieve good model performance. This is due to the fact that each client only contributes a small fraction of the data in each round.

Federated Stochastic Gradient Descent is a direct adaptation of the traditional SGD algorithm [1] to the federated setting. In the FedSGD process, the server starts by initializing the global model weights w_t. For each communication round t, a fraction C of clients is randomly selected to participate. Each selected client k uses its local dataset $\mathcal{P}_k$ to compute the gradient of the loss function $f_i(w)$ over all its data points n_k. These gradients G_k are then sent to the central server, where they are aggregated. The server updates the global model weights w_t by taking a weighted average of the received gradients, with weights proportional to the number of data points n_k on each client. The updated global model w_{t+1} is then distributed back to all clients, and the process repeats until the stopping criterion is met.

ALGORITHM 3.1: *Federated Stochastic Gradient Descent (FedSGD)*

Server Execution:
Initialize global model weights w_0, and learning rate η
for each round $t = 1, 2, \ldots$ **do**
    $m \leftarrow \max(C \cdot K, 1)$, number of participating clients
    $S_t \leftarrow$ random set of m clients
    Send global model w_t to S_t clients
    for each client $k \in S_t$ **in parallel do**
        $g_k \leftarrow$ GRADIENTSTEP(k, w_t)
    end for
    $w_{t+1} \leftarrow w_t - \eta \sum_{k \in S_t} \frac{n_k}{n} g_k$
    Send the model w_{t+1} to all participants
end for

Client Execution:
procedure GRADIENTSTEP(k, w_t)
    $g \leftarrow \nabla f_i(w_t)$ over $\mathcal{P}_k$
    return g to server
end procedure

3.3.2 Federated Averaging (FedAvg)

While FedSGD provides a straightforward extension of SGD to the federated setting, it requires frequent communication between clients and the server, which can be a significant bottleneck in environments with limited bandwidth. To address this issue, the Federated Averaging (FedAvg) algorithm was proposed, introducing several key innovations to reduce communication overhead while maintaining model performance.

FedAvg is an extension of the FedSGD approach, allowing clients to perform multiple local updates using their own data before communicating with the server. Essentially, this means that each client performs several local gradient descent steps on their model using local data. After these local updates, clients communicate their updated model parameters, rather than a single

[1] FedSGD resembles the logic of Batch Gradient Descent, where the gradients over the entire dataset is computed and then aggregated by the Server.

gradient, to the server. These updates are subsequently aggregated by the server. This approach not only reduces the frequency of communication rounds but also leverages local computations more effectively, particularly considering the ever-increasing performance of contemporary IoT devices used as clients.

ALGORITHM 3.2: *Federated Averaging (FedAvg)*

Server executes:
Initialize global model weights w_0, and learning rate η
for each round $t = 1, 2, \ldots$ **do do**
    $m \leftarrow \max(C \cdot K, 1)$
    $S_t \leftarrow$ random set of m clients
    Send global model w_t to S_t clients
    for each client $k \in S_t$ **in parallel do do**
        $w_t^k \leftarrow \text{ClientUpdate}(k, w_t)$
    end for
    $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_t^k$
end for

ClientUpdate(k, w_t):
$B \leftarrow$ split $\mathcal{P}_k$ into batches of size B
for each local epoch i from 1 to E **do do**
    for batch $b \in B$ **do do**
        $w \leftarrow w - \eta \nabla \ell(w; b)$
    end for
end for
return w to server

All in all, the amount of computation can be adjusted using three key learning parameters: C, the fraction of clients participating in each round; E, the number of local epochs or training passes each client performs over its local dataset per round; and B, the local minibatch size used for client updates. In the extreme case where $B \to \infty$ and $E = 1$, FedAvg naturally simplifies to FedSGD, thereby concealing the unique novelties of FedAvg.

The novelties introduced by FedAvg, briefly outlined above, made the algorithm a more practical and efficient approach to federated learning, addressing some of the key limitations of FedSGD. To sum up, these innovations include reduced communication frequency, local computation utilization, and improved scalability. By performing multiple local updates before sending the model to the server, FedAvg significantly reduces the number of communication rounds required. Concurrently, clients leverage their local computational resources more effectively, due to the increasing processing power of modern edge devices. Finally, taking load out of the server with the local updates provides the foundations for enhanced scalability as the processing load is distributed.

3.3.3 Contempory Challenges of Federated Learning

Federated Learning (FL) faces several contemporary challenges that need to be addressed to enhance its implementation and efficiency. The objective of this research was, among others, to provide a pathway on how to overcome some of these issues, yet the entirety of them would be impossible to be resolved solely with one thesis dissertation. In this spirit, to be able to illustrate succinctly the various shortcomings of Federated Learning, whilse highlighting their relevance to this research study, an outline of the principal challenges, along with insights into potential solutions that are presented in our implementation will be presented. The following analysis is based on the journal article introduced by Li et al.[14] (2020):

Complex infrastructure and Management: Federated Learning demands a robust and complex infrastructure for coordinating numerous distributed devices with diverse hardware, software, and network conditions, making setup and maintenance challenging. Ensuring seamless synchronization, managing device failures, and handling system heterogeneity require sophisticated solutions. Additionally, secure data transmission and efficient resource allocation are essential. Despite advances in machine learning infrastructure and research, the complexity of FL infrastructure reveals a need for simplified deployment and management interfaces. This gap highlights the necessity for easy federation of machine learning processes, making FL more accessible and manageable.

Expensive Communication: Communication Overhead is a critical bottleneck in Federated Networks, that necessitates against transmitting locally generated data. The reality in Federated Learning setups differs significantly from the almost perfect communication setups of traditional Centralized Learning data centers, where stable and incredibly fast connections are assumed with restricted failures. In contrast, Federated Learning is conducted in an outright opposite manner, meaning that learning is based on utilizing a vast number of devices, like smartphones, resulting in substantially reduced communication capabilities with several limiting factors including bandwidth, energy and transmission power. Therefore, in edge FL, where the datasets are small and the connections are weak and unreliable the communication-to-computation ratio skyrockets.

System Heterogeneity : Devices in federated networks exhibit significant variability in storage, computational power, and communication capabilities due to differences in hardware (CPU, memory), network connectivity (3G, 4G, 5G, Wi-Fi), and power levels (battery). Typically, only a small fraction of devices are active at any given time, and devices may drop out due to connectivity or energy constraints. These system-level disparities aggravate challenges such as managing stragglers, who delay updates, and ensuring fault tolerance within the network. Robust methods are needed to handle low participation and heterogeneous environments effectively.

Statistical Data Heterogeneity: The need for independent and identically distributed (i.i.d.) datasets is crucial for efficiency in Distributed Systems. In Federated Systems, while non-i.i.d. data can be used, it significantly impacts the speed and difficulty of convergence. Each client trains on its local dataset, adding complexity and likely resulting in skewed updates based on user preferences. The server, due to privacy concerns, cannot know the dataset distribution, necessitating sophisticated approaches to maintain accuracy while learning from highly biased datasets. These methods must effectively address the challenges posed by data heterogeneity to ensure efficient federated learning.

Privacy and Security: The main objective of Federated Learning is to ensure client privacy. However, shared parameters can be exploited by malicious participants or third parties to extract sensitive information, undermining this goal. It is commonly assumed that all clients act in good faith, but some may be malicious, attempting to compromise the model's integrity. They can perform model poisoning or data poisoning attacks, using corrupted data to degrade model accuracy or introduce backdoors. Therefore, robust security measures are crucial to maintain the integrity and privacy of the FL process.

3.3.4 Countermeasures Introduced in Thesis

This thesis addresses several key challenges in Federated Learning, some as primary objectives and others as secondary considerations. The following discussion is organized by relevance to

the thesis contents, starting with the main objectives: mitigating communication overhead and providing a user-friendly interface for FL experimentation.

To reduce communication overhead in FL, this work employs strategies such as minimizing the size of transmitted messages, reducing the number of communication rounds, and limiting communication frequency. Specifically, the thesis implements the Functional Dynamic Averaging (FDA) synchronization technique (section 3.4) to enhance convergence speed, effectively balancing the trade-offs between communication and computation. Additionally, the Flower Framework (section 4.2) is utilized to facilitate a seamless transition from traditional ML frameworks to FL setups, offering an accessible and efficient infrastructure.

While the issue of system heterogeneity is not directly addressed, the thesis integrates a Federated Learning monitoring framework with the NS3 network simulator (section 4.3). This integration enables experimentation with various communication setups. Additionally, the framework monitors data heterogeneity to assess system performance under different data distribution conditions.

Privacy and security concerns, although not directly tackled in this study, are acknowledged as critical areas for future research. The experiments are conducted in a controlled environment, which minimizes the immediate need for additional privacy measures. However, addressing these concerns remains essential for the advancement of Federated Learning.

3.4 Synchronization Techniques

Implementing a Federated Learning (FL) system presents unique synchronization challenges, essential for maintaining model accuracy and consistency across distributed nodes. Synchronization is not only critical for synchronous learning tasks, where synchronized model distribution and aggregation are mandatory (see section 2.5.4), but also for any multi-threaded process, including both synchronous and asynchronous setups. Effective synchronization in FL is vital to coordinate the operations of multiple distributed devices, handle threads execution, and ensure the consistency and accuracy of the global model. This research emphasizes synchronous Distributed Learning scenarios, necessitating careful considerations to manage model aggregation and ensure thread safety. To achieve this, various synchronization techniques are employed, focusing on both Synchronous Federated Learning methods and traditional synchronization primitives used in multiprogramming environments.

3.4.1 Geometric Method

The Geometric Method comprises an innovative approach for effectively overseeing threshold functions across distributed data streams introduced by Sharfman et al. (2010)[17] with their research "A Geometric Approach to Monitoring Threshold Functions over Distributed Data Streams". The primary motivation of the proposed method is to mitigate the communication costs entailed by frequent interprocess communication— a common characteristic of distributed learning processes. In previous sections, it was highlighted that in order to tackle communication overheads, the two major approaches are either to restrict the size of the communicated information or to minimize the frequency of communication. The geometric reformulation of the issue at hand, drastically reduces the need for communication in distributed scenarios, enlisting the aforementioned technique under the latter category.

The communication frequency restriction is accomplished through the decomposition of the global monitoring task into a set of localized constraints, derived from the geometric properties of the problem. In this manner, worker nodes are enabled to individually monitor their data

streams, utilizing outsourced communication with the parameter server or other workers solely on the occasion that these local constraints are violated. Such violations indicate that the local data changes are significant enough to potentially affect the global monitored function, while irrelevant data changes are filtered out.

Overview of Implementation:

Having established a foundational basis of the general concept introduced by the geometric method, at this point, we will delve further into some key aspects regarding the implementation of the underlying logic. Consider a network comprised by n nodes and $\mathbb{S} = \{s_1, s_2, \ldots, s_n\}$ being an arbitrary set of data streams collected by nodes of $\mathbb{P} = \{p_1, p_2, \ldots, p_n\}$ representing the set of nodes. Then we can define two types of vectors crucial for the analysis:

$$\begin{cases} u_1(t), u_2(t), \ldots, u_n(t) & \text{: Local statistics vectors} \\ u(t) = \frac{\sum_{i=1}^{n} w_i u_i(t)}{\sum_{i=1}^{n} w_i} & \text{: Global statistics vector} \end{cases} \tag{3.1}$$

Where in the above statements $u_i(t) \in \mathbb{R}^d$, represents the local statistics of node p_i and w_i is a positive weight assigned to node p_i, usually corresponding to the population of elements in the local statistics of the node. Combined in the second equation we get the weighted average of local statistics producing a global illustration.

Nonetheless, in order to maintain the value of the *global statistics* $(u(t))$, at each time t the *local statistics vectors* $(u_i(t))$ would need to be transmitted, resulting in expensive communications. Thus, another vector is introduced denoted as $e(t)$ that represents an estimate of the global statistics and is computed by the *local statistics vectors* collected at certain times. This *estimate vector* is known at any time by all the nodes. Before presenting the mathematical definition u_i' needs to be introduced, which represents the last statistics vector collected by the node p_i:

$$e(t) = \frac{\sum_{i=1}^{n} w_i u_i'}{\sum_{i=1}^{n} w_i}$$

Therefore the estimate vector essentially is the weighted average of the latest statistics vectors collected from the nodes. Finalizing with the notation: the *statistics delta vector:* $\Delta u_i(t)$ and the *the drift vector:* $v_i(t)$ are the last essential vectors to define:

$$\begin{cases} \Delta u_i(t) = u_i(t) - u_i' & \text{: Statistics delta vector} \\ v_i(t) = e(t) + \Delta u_i(t) & \text{: Drift vector} \end{cases} \tag{3.2}$$

The first portrays the difference between the current local statistics vector and the last statistics vector transmitted and the second displacement of $\Delta u_i(t)$ in relation to the estimate vector.

It is important to note at this point that the geometric method finds application in two types of environments. That way the implementation branches in a decentralized setting, where nodes can efficiently broadcast messages, and a coordinator-based setting, where the broadcasting cost is high. The variables previously defined are relevant for both setups, yet the process slightly differs. At this point, an overview of the algorithm will be provided in both setups to attain a better illustration of the process. Subsequently, the undelaying logic will be displayed to form an understanding of the geometric method.

Decentralized Setting:

In the decentralized setting, when the algorithm dictates that a node's local statistics vector needs to be transmitted, meaning it is important to update the estimate vector to maintain accu-

racy, then that same node broadcasts his local statistics vector to the rest of the nodes. In this type of setting, all nodes retain in memory the last vector broadcasted by the rest (u'_i for i $\in [1, n]$), and when an update arrives from node p_j then u'_j is set to the new value, and the estimate vector is recalculated. The algorithmic implementation is provided in algorithm 3.3.

ALGORITHM 3.3: *The Decentralized Algorithm of the Geometric Approach*

INITIALIZATION: AT A NODE p_i:
- Broadcast a message containing the initial statistics vector and update v'_i to hold the initial statistics vector. Upon receipt of messages from all the nodes, calculate the estimate vector ($e(t)$).

PROCESSING STAGE AT A NODE p_i:
- Upon the arrival of new data on the local stream, recalculate $v_i(t)$, and $u_i(t)$, and check if $B(e(t), u_i(t))$ remains monochromatic. If not, broadcast the message $< i, v_i(t) >$ and update v'_i to hold $v_i(t)$.
- Upon receipt of a new message $< j, v_j(t) >$, update v'_j to hold $v_j(t)$, recalculate $e(t)$, and check if $B(e(t), u_i(t))$ is monochromatic. If $B(e(t), u_i(t))$ is not monochromatic, broadcast the message $< i, v_i(t) >$ and update v'_i to hold $v_i(t)$.

Coordinator-based Setting:

In the coordinator-based setting, a coordinator node is designated that assumes responsibility for collecting the local statistics vectors from the nodes, to calculate the estimate vector to distribute it to the rest. This way there are no more expensive broadcast operations, and this is a task completed by the coordinator. In addition to this modification, another element of the coordinator-based architecture is that it employs a mechanism for balancing the local statistic vectors of a subset of nodes. To achieve this, the second equation of eq. (3.2) is slightly modified to look like the following:

$$v_i(t) = e(t) + \Delta u_i(t) + \frac{\delta_i}{w_i} \tag{3.3}$$

with this adjustment, a so-called *slack vector* (δ_i) is additionally sent from the coordinator when it is observed that two nodes p_i and p_j cancel each other out: $\Delta_i(t) = -\Delta_j(t)$. The total sum of the slack vectors sent to the nodes is 0, that way in a global perspective the estimate statistics vector is not influenced, yet the reinstated values of the nodes drift vector lead to better performance.

Unlike the Decentralized algorithm, the coordinator-based counterpart provides a more sophisticated approach that is preferable to be explained swiftly instead of providing a complicated algorithmic implementation.

The coordinator-based algorithm involves the central coordinator managing the local statistics and slack vectors from distributed nodes to monitor a global threshold function efficiently. The process starts with each node sending its initial statistics vector to the coordinator, which then calculates an estimate vector and broadcasts it to all nodes. As new data arrives, nodes locally update their statistics and drift vectors and check if their local constraints are violated. If a violation occurs, the node sends a report to the coordinator. The coordinator attempts to balance the system by adjusting slack vectors among a subset of nodes to maintain the correctness of the estimate vector. If balancing fails, a new estimate vector is calculated and broadcasted. This way communication overhead is reduced by restricting the interactions between the coordinator and a subset of nodes rather than frequent broadcasts from one node to the rest.

Geometric Property:

To form an understanding of the two presented algorithms and grasp the geometric connection, consider the task of decomposing a global monitoring task into local constraints on data streams. Each node p_i verifies if its local constraint has been violated. If none of these constraints are violated, the query result remains unchanged, requiring no communication.

The global statistics vector $u(t)$ at time t is the weighted average of the drift vectors $v_i(t)$ held by the nodes:

$$u(t) = \frac{\sum_{i=1}^{n} w_i v_i(t)}{\sum_{i=1}^{n} w_i} \tag{3.4}$$

This implies that the global statistics vector $u(t)$ is within the convex hull of the drift vectors:

$$u(t) \in \text{Conv}(v_1(t), v_2(t), \ldots, v_n(t)) \tag{3.5}$$

To monitor the local constraints, we use Theorem 1, which states that for vectors $x, y_1, y_2, \ldots, y_n \in \mathbb{R}^d$, the convex hull $\text{Conv}(x, y_1, y_2, \ldots, y_n)$ can be bounded by n d-dimensional balls $B(x, y_i)$ each centered at $\frac{x+y_i}{2}$ with radius $\frac{\|x-y_i\|}{2}$:

$$\text{Conv}(x, y_1, y_2, \ldots, y_n) \subseteq \bigcup_{i=1}^{n} B\left(\frac{x + y_i}{2}, \frac{\|x - y_i\|}{2}\right) \tag{3.6}$$

In our scenario, we construct n balls $B(e(t), v_i(t))$ where each ball is centered at $\frac{e(t)+v_i(t)}{2}$ with radius $\frac{\|e(t)-v_i(t)\|}{2}$, ensuring that:

$$\text{Conv}(e(t), v_1(t), v_2(t), \ldots, v_n(t)) \subseteq \bigcup_{i=1}^{n} B\left(\frac{e(t) + v_i(t)}{2}, \frac{\|e(t) - v_i(t)\|}{2}\right) \tag{3.7}$$

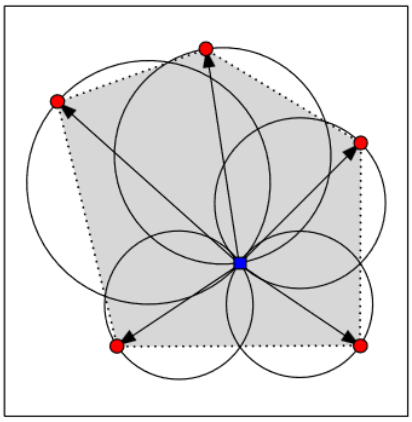


Figure 3.7. *Drift vectors comprising the convex hull[17]*

Each node checks if the ball $B(e(t), v_i(t))$ is monochromatic. If a ball is monochromatic red, $\{x \mid f(x) < r\}$, no communication is required. If any ball contains green vectors, $\{y \mid f(y) \geq r\}$, the node reports its local statistics, indicating a local constraint breach. Figure 3.7 illustrates this concept, showing the drift vectors $v_i(t)$ for a 2-dimensional statistic on a plane. The convex hull of the drift vectors is shown in gray, and the union of the balls $B(e(t), v_i(t))$ encompasses this convex hull, allowing nodes to collectively monitor the global constraint.

In conclusion, the geometric method presented a substantial breakthrough in monitoring threshold functions over distributed data streams. The findings of this work provided a pathway for implementing functional monitoring mechanisms, in distributed systems, crucial for the

synchronization of Distributed Learning (DL) processes, and by extension Federated Learning (FL). These mechanisms can substantially reduce the frequency of communication and achieve efficient learning by applying some simple redefinitions to fit our problem's requirements. Specifically, replacing *nodes* with *clients*, *statistics* with *model parameters*, and providing a fitting *monitor function* of the *local statistics* presents a methodology directly applicable to the federated setting at hand.

3.4.2 Functional Dynamic Averaging

The Groundwork:

The encouraging findings of the geometric method[17], propelled further research leading to the publication of "Sketch-based Geometric Monitoring of Distributed Stream Queries"[80] M.Garofalalakis, V.Samoladas and D.Keren with the first two proceeding with "Functional Geometric Monitoring for Distributed Streams" [18].

Sketch-based Geometric Monitoring[80] extends the geometric method concept by integrating AMS (Alon-Matias-Szegedy) sketches[81] with geometric monitoring, enabling efficient and scalable monitoring of complex, non-linear aggregate queries. This combination allows for significant dimensionality reduction, providing error bounds and approximation guarantees, and further reducing communication costs, making the method robust against high variability and skew in data streams.

Additionally, the subsequent introduction of Functional Geometric Monitoring (FGM) comprised a conjunction of the fundamental ideas of geometric monitoring[17] (GM) and the sketches based approach[80].

Functional Geometric Monitoring (FGM) employs non-linear functions to project local statistics vectors, or summary vectors, enhancing the flexibility and performance of monitoring even in scenarios with high data variability and skewness. This approach further reduces communication costs by maintaining performance within adaptable and strict bounds, closely approximating the coordinator-based version of GM (see section 3.4.1). FGM also achieves a clear delineation between monitoring logic and distributed system issues, providing a viable solution for the monitoring logic of any general-purpose middleware platform. Additionally, FGM introduces the utilization of sketches to approximate local summary vectors and incorporate them into the geometric monitoring framework, enabling accurate monitoring of complex distributed stream queries.

Methodology:

Building on the promises of previously explored synchronization methods, Functional Dynamic Averaging (FDA) was introduced by V.Konidaris and V.Samoladas [16] as a sophisticated synchronization strategy for coordinator-based distributed architectures. FDA presents a compelling approach that can be modified to fit any Distributed or Federated Learning architecture that requires synchronization as it adopts the characteristic of FGM for separation of the learning process with the undelaying monitoring logic. The application of the aforementioned approach resulted in substantial mitigation of communication in distributed environments accomplishing not only swift training convergence but also high levels of accuracy, ranking higher than comparable synchronization methods developed in the same context.

The proposed approach aims to create a global monitoring task that ensures only significant changes in local models are reported, thereby preserving network bandwidth and reducing latency. Similar to Geometric Monitoring (GM) logic, the objective is to approximate the global monitoring task by a set of local constraints that can be monitored by all clients. A fundamental principle of Distributed Machine Learning (DML) is that the global model maintains a good perception of the obtained knowledge, resulting in better performance when it is closely approximated by the

weighted average of all local models. Thus, the proposed constraints must guarantee this initial condition.

The global model is notated as $\bar{w}$ and is generally unknown throughout the learning process. Let $w_t^{(k)}$ be the the model of client k at moment t, then the global model is calculated as follows:

$$\bar{w} = \sum_{k=1}^{n} \frac{n_k}{n} w_t^{(i)} \tag{3.8}$$

This indicates, however, that at any given moment t the clients need to be transmitting their model to the server. Such an implementation would be unreasonable as it would incur particularly frequent communications, deteriorating the performance. Naturally, to address this issue FDA, similar to other synchronization methods (including BSP, TAP, and SSP), works in rounds, and when the end of a specific round is signaled then the perceived global model is updated. In other words, at the end of a learning round, all clients send their local models to the server for aggregation as dictated by eq. (3.8). The newly-calculated global model is redistributed to the clients providing an estimate of the global model.

Nonetheless, the question that needs to be answered is how the end of a round is determined. The method of FDA applies a principle that stems from the FGM approach, utilizing a non-linear projection function to transform the local statistics-summary vectors into real numbers, which are collectively monitored. In FDA the role of the projection function is assumed by the variance of the local model in comparison with the global estimate. The reason why variance, is selected is to measure how accurate is the perception we maintain for the global model. In the case that local models start deviating substantially from the global estimate, this means that a global update is required to bridge this gap, or else the global model's accuracy will deteriorate.

The reason the above is important to answer the question posed is that, in order to identify the end of a round, the variance value needs to surpass a threshold value Θ. Θ is a hyperparameter of Functional Dynamic Averaging (FDA) defined at the start of a training round and can be subsequently reconfigured based on the requirements of the synchronization algorithm. The mathematical representation of the above is as follows:

$$\frac{1}{n} \sum_{k=1}^{n} \left\| \mathbf{w}_t^{(k)} - \overline{\mathbf{w}}_t \right\|^2 \leq \Theta \tag{3.9}$$

The condition depicted above is the most crucial part of FDA and is named the Round Termination Condition (RTC), with the left part presenting the average model variance and the right the threshold Θ. The fraction outside is due to the fact that clients proceed synchronized leading to equal amount of data stemming from each, thus resulting in equal contributions. As long as the RTC is not violated, it can be inferred that the global estimate of the model is accurate and performs efficiently.

FDA introduces three major approaches—Naive FDA, Linear FDA, and Sketch FDA—differing mainly in how they monitor the Round Termination Condition (RTC) and approximate the variance of the local models. Each method takes a different approach to balancing the trade-off between communication overhead and the accuracy of synchronization.

RTC Monitoring:

As stated previously in chapter 5, The underlying technique of RTC monitoring is entailed by the fundamental principles of FGM[18]. In this context, Assume there are n clients in the learning network. Each of these clients has a local stream generated locally or collected. Let $S_k(t) \in \mathbb{R}^m$ denote the *local state vector* of client k in moment t. The *global state vector* can be denoted as

$S(t) \in \mathbb{R}^m$ and is computed as the average of all local states.

$$S(t) = \frac{1}{k} \sum_{i=1}^{k} S_i(t) \tag{3.10}$$

The goal is to monitor a bounded condition on the global model, by projecting the global function into real numbers and comparing the projection with threshold Θ. The projection function is non-linear and is denoted as $F : \mathbb{R}^m \to \mathbb{R}$ leading to the following condition:

$$F(S(t)) \leq \Theta \overset{3.10}{\iff} F(\frac{1}{k} \sum_{i=1}^{k} S_i(t)) \leq \Theta \tag{3.11}$$

Additionally, let t_0 be the time when the latest communication round started. At this initial time, the local models are synchronized, so $w_{t_0}^{(1)} = w_{t_0}^{(2)} = \cdots = w_{t_0}^{(n)} = \bar{w}_{t_0}$. The *update of learner* k at step t is the difference between the weights at step t and the weights at the last synchronization t_0 which represents the estimation of the global state. Thus the average update can also be induced as the mean of the clients' updates. Overall:

$$\begin{cases} \Delta_t^{(k)} = w_t^{(k)} - w_{t_0}^{(k)}\text{: Update of learner } k \text{ at step } t \\ \bar{\Delta}_t = \frac{1}{n} \sum_{k=1}^{n} \Delta_t^{(k)}\text{: Average update} \end{cases} \tag{3.12}$$

With these definitions in mind, we can proceed and reformulate the norm calculation on the left side of RTC eq. (3.9):

$$\begin{aligned} \frac{1}{n} \sum_{k=1}^{n} \left\| w_t^{(k)} - \bar{w}_t \right\|^2 &= \frac{1}{n} \sum_{k=1}^{n} \left\| \left(w_t^{(k)} - w_{t_0}^{(k)} \right) - \left(\bar{w}_t - w_{t_0}^{(k)} \right) \right\|^2 \\ &= \frac{1}{n} \sum_{k=1}^{n} \left\| \Delta_t^{(k)} - \bar{\Delta}_t \right\|^2 \\ &= \frac{1}{n} \sum_{k=1}^{n} \left(\left\| \Delta_t^{(k)} \right\|^2 - 2\Delta_t^{(k)} \cdot \bar{\Delta}_t + \left\| \bar{\Delta}_t \right\|^2 \right) \\ &= \left(\frac{1}{n} \sum_{k=1}^{n} \left\| \Delta_t^{(k)} \right\|^2 \right) - 2 \left(\frac{1}{n} \sum_{k=1}^{n} \Delta_t^{(k)} \right) \cdot \bar{\Delta}_t + \left\| \bar{\Delta}_t \right\|^2 \\ &= \left(\frac{1}{n} \sum_{k=1}^{n} \left\| \Delta_t^{(k)} \right\|^2 \right) - \left\| \bar{\Delta}_t \right\|^2 \end{aligned} \tag{3.13}$$

Thus, conceptually we can set the local state of client k and the function F in the following manner to satisfy our objective eq. (3.11):

$$\begin{cases} S_k(t) = \left[\|\Delta_t^{(k)}\|^2, \bar{\Delta}_t \right]^T \in \mathbb{R}^{m+1}, \\ F\left([v, x]^T \right) = v - \|x\|^2 \end{cases} \tag{3.14}$$

Using these aforementioned definitions, the RTC can be expressed as:

$$F(S(t)) \leq \Theta,$$

Practical Considerations

Defining the local state $S_k(t)$ in the described manner in eq. (3.14) can be resource-intensive for the network, as the local update $\Delta_t^{(k)}$, required for the computation of $\bar{\Delta}_t$ (refer to eq. (3.12)), has m dimensions, corresponding to the number of weights in the local client's model. Therefore,

our goal is to examine different ways to define the local states and the function F to minimize network overhead. This process leads to the presentation of the three proposed FDA approaches, each opting for a different method of dimensionality reduction on the local updates $\Delta_t^{(k)}$. The idea is that as the approximation of the actual RTC becomes stricter, the cost of communication increases. This trade-off is inevitable and will be studied further, with the three methods being presented in order of increasing communication cost and decreasing approximation accuracy.

Nonetheless, at this point, the general algorithmic approach of FDA is attached to maintain a common offset before the upcoming branching in the three approaches. The following algorithm is a product of M. Theologitis thesis-dissertation[82].

ALGORITHM 3.4: *FDA: Federated Learning training with Approximate RTC Monitoring*

Require: $f(w)$: Stochastic objective function with parameters $w \in \mathbb{R}^d$
Require: K: The number of clients indexed by k
Require: Θ: Monitoring threshold
Require: $S_k(t)$: The local state for client k where $S_k(t) \in \mathbb{R}^p$ and $p \ll d$
Require: $F(x)$: A function $F : \mathbb{R}^p \to \mathbb{R}$ such that $F(S(t)) \leq \Theta$ implies the RTC
Require: η: Step size (learning rate)
Require: s: The number of local steps
Require: b: The local mini-batch size
Server executes:
initialize w_1
for each round $t = 1, 2, \ldots$ **do**
Broadcast w_t to all clients
repeat
for each client $k = 1, \ldots, K$ **in parallel do**
$S_k(t) \leftarrow$ ClientTrain(k)
end for
until $F(S(t)) > \Theta$
for each client $k = 1, \ldots, K$ **in parallel do**
$w_{t+1}^{(k)} \leftarrow$ (download the model of client k)
end for
$w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^{(k)}$
end for
ClientTrain(k):
$\mathcal{B} \leftarrow$ (Choose s batches of size b from $\mathcal{P}_k$)
for each batch $B \in \mathcal{B}$ **do**
$w \leftarrow w - \eta \nabla f_B(w)$
end for
return $S_k(t)$ to server

Naive Approximation

The Naive FDA approach ignores the local state vector and simplifies the problem by reducing the local state to a single scalar value, specifically the squared norm of the local update vector. This approach is straightforward and incurs minimal computation overhead, making it suitable for scenarios where communication costs are critical. The local state for client k at time t, along with the projection function F, are redefined as follows:

$$\begin{cases} S_k(t) = \|\Delta_t^{(k)}\|_2^2 \\ F(v) = v \end{cases} \tag{3.15}$$

The function $F(v) = v$ is used to aggregate the states across clients. The RTC is checked using the condition $F(S(t)) \leq \Theta$. The proof that Naive FDA satisfies the RTC involves starting from the

condition we want to prove and with logical equalities conclude in a known condition (RTC):

$$
\begin{aligned}
F(S(t)) \leq \Theta &\iff \frac{1}{n}\sum_{k=1}^{n}\left\|\Delta_t^{(k)}\right\|^2 \leq \Theta \\
&\overset{\left\|\overline{\Delta}_t\right\|^2 \geq 0}{\iff} \frac{1}{n}\sum_{k=1}^{n}\left\|\Delta_t^{(k)}\right\|^2 - \left\|\overline{\Delta}_t\right\|^2 \leq \frac{1}{n}\sum_{k=1}^{n}\left\|\Delta_t^{(k)}\right\|^2 \leq \Theta \\
&\overset{3.13}{\iff} \frac{1}{n}\sum_{k=1}^{n}\left\|\mathbf{w}_t^{(k)} - \overline{\mathbf{w}}_t\right\|^2 \leq \frac{1}{n}\sum_{k=1}^{n}\left\|\Delta_t^{(k)}\right\|^2 \leq \Theta \\
&\implies \frac{1}{n}\sum_{k=1}^{n}\left\|\mathbf{w}_t^{(k)} - \overline{\mathbf{w}}_t\right\|^2 \leq \Theta \quad \underline{\text{RTC}} \quad (\textit{Equation (3.9)})
\end{aligned}
\tag{3.16}
$$

Since this approximation overestimates the actual variance, ensuring $F(S(t)) \leq \Theta$ implies that the RTC is met. This approach is the most simple requiring merely a 1-dimensional scalar value to be communicated to monitor RTC. In most cases, accuracy is not heavily impacted, yet this approximation is based on a loose condition, meaning that deviation from the required results can be noticed in some cases.

3.4.3 Linear Approximation

The Linear FDA method provides a more refined approach by reducing the local update vector $\Delta_t^{(k)}$ to a scalar product with a unit vector ξ, meaning $\xi \cdot \Delta_t^{(k)} \in \mathbb{R}$. This method balances computational efficiency and accuracy in estimating the variance of the updates. The local state for client k at time t, along with the projection function F, are defined as:

$$
\begin{cases}
S_k(t) = \left[|\Delta_t^{(k)}|_2^2, \langle \xi, \Delta_t^{(k)} \rangle\right]^\top \\
F(v, x) = v - x^2
\end{cases}
\tag{3.17}
$$

The function $F(v, x) = v - x^2$ is used to aggregate the states across clients. The RTC is checked using the condition $F(S(t)) \leq \Theta$. The proof that Linear FDA satisfies the RTC follows the same methodology as in Naive:

$$
\begin{aligned}
F(S(t)) \leq \Theta &\iff \frac{1}{n}\sum_{k=1}^{n}\left\|\Delta_t^{(k)}\right\|^2 - \frac{1}{n}\sum_{k=1}^{n}(\xi \cdot \Delta_t^{(k)})^2 \leq \Theta \\
&\overset{\xi^2=1}{\iff} \frac{1}{n}\sum_{k=1}^{n}\left\|\Delta_t^{(k)}\right\|^2 - \left\|\overline{\Delta}_t\right\|^2 \leq \frac{1}{n}\sum_{k=1}^{n}\left\|\Delta_t^{(k)}\right\|^2 - (\xi \cdot \overline{\Delta}_t)^2 \leq \Theta \\
&\overset{3.13}{\iff} \frac{1}{n}\sum_{k=1}^{n}\left\|\mathbf{w}_t^{(k)} - \overline{\mathbf{w}}_t\right\|^2 \leq \frac{1}{n}\sum_{k=1}^{n}\left\|\Delta_t^{(k)}\right\|^2 - (\xi \cdot \overline{\Delta}_t)^2 \leq \Theta \\
&\implies \frac{1}{n}\sum_{k=1}^{n}\left\|\mathbf{w}_t^{(k)} - \overline{\mathbf{w}}_t\right\|^2 \leq \Theta \quad \text{RTC (Equation (3.9))}
\end{aligned}
\tag{3.18}
$$

This method ensures accurate variance estimation, with the condition $F(S(t)) \leq \Theta$ indicating the RTC is met. It balances communication efficiency and accuracy, making it suitable for scenarios requiring a more precise estimate than the Naive method. Notably, randomly selecting ξ can lead to inadequate performance and early termination, as it may be nearly orthogonal to Δ. A preferable choice is a vector correlated with Δ. One heuristic is to use Δ_0 (normalized to unit norm), the update vector from just before the current round. Each node can estimate this independently as the difference between the last two models, $\tilde{w}_0 - \tilde{w}_1$, sent by the coordinator.

3.4.4 Sketch FDA

The Sketch FDA method utilizes AMS sketches[80] to compress local update vectors, achieving great balance between communication efficiency and accuracy. This technique is particularly advantageous in scenarios involving high-dimensional data that require significant dimensionality reduction[83].

The AMS sketch denoted as $\text{sk}(\mathbf{v})$, is a technique that compresses a high-dimensional vector into a significantly smaller matrix. For a vector $\mathbf{v} \in \mathbb{R}^M$, the sketch $\text{sk}(\mathbf{v})$ is defined as:

$$\text{sk}(\mathbf{v}) = \Xi = [\xi_1 \quad \xi_2 \quad \dots \quad \xi_d], \tag{3.19}$$

where each ξ_i is a sub-vector of dimension d, and typically $d \cdot m \ll M$.

The sketch function is linear and can be computed in $O(dM)$ steps. Furthermore, there exists a function, denoted as $\mathcal{M}_2$, that accurately estimates the squared norm of a vector $\mathbf{v}$ using its sketch $\text{sk}(\mathbf{v})$. This function is defined as:

$$\mathcal{M}_2(\text{sk}(\mathbf{v})) = \text{median}_{i=1,\dots,d}\|\xi_i\|^2. \tag{3.20}$$

For $m = O\left(\frac{1}{\epsilon^2}\right)$ and $d = O\left(\log \frac{1}{\delta}\right)$, it is proven that with high probability $(1-\delta)$, the squared norm $\|\mathbf{v}\|^2$ can be approximated within a factor of $(1-\epsilon)$ to $(1+\epsilon)$:

$$(1-\epsilon)\|\mathbf{v}\|^2 \leq \mathcal{M}_2(\text{sk}(\mathbf{v})) \leq (1+\epsilon)\|\mathbf{v}\|^2. \tag{3.21}$$

In our context, the large vector requiring compression is the local update $\Delta_t^{(k)} \in \mathbb{R}^M$. Consequently, the local state $S_k(t)$ and the function F for the sketch method are defined as follows:

$$\begin{cases} S_k(t) = \left[\left\|\Delta_t^{(k)}\right\|^2 \quad \text{sk}\left(\Delta_t^{(k)}\right)\right]^T \in \mathbb{R}^{1+d\times m}, \\ F\begin{pmatrix} v \\ \Xi \end{pmatrix} = v - \frac{1}{1+\epsilon}\mathcal{M}_2(\Xi). \end{cases} \tag{3.22}$$

To establish that the RTC holds, we proceed as follows:

$$\begin{aligned} F(S(t)) \leq \Theta \iff & \frac{1}{n}\sum_{k=1}^{n}\left\|\Delta_t^{(k)}\right\|^2 - \frac{1}{1+\epsilon}\cdot\frac{1}{n}\sum_{k=1}^{n}\mathcal{M}_2\left(\text{sk}\left(\Delta_t^{(k)}\right)\right) \leq \Theta \\ \overset{\text{linearity}}{\iff} & \frac{1}{n}\sum_{k=1}^{n}\left\|\Delta_t^{(k)}\right\|^2 - \frac{1}{1+\epsilon}\cdot\mathcal{M}_2\left(\text{sk}\left(\Delta_t\right)\right) \leq \Theta \\ \overset{3.21}{\iff} & \frac{1}{n}\sum_{k=1}^{n}\left\|\Delta_t^{(k)}\right\|^2 - \left\|\overline{\Delta}_t\right\|^2 \leq \frac{1}{n}\sum_{k=1}^{n}\left\|\Delta_t^{(k)}\right\|^2 - \frac{1}{1+\epsilon}\mathcal{M}_2\left(\text{sk}\left(\Delta_t\right)\right) \leq \Theta \\ \overset{3.13}{\iff} & \frac{1}{n}\sum_{k=1}^{n}\left\|\mathbf{w}_t^{(k)} - \overline{\mathbf{w}}_t\right\|^2 \leq \Theta \quad \underline{\text{RTC}} \text{ (Equation (3.9))} \end{aligned} \tag{3.23}$$

3.4.5 Multiprogramming & Synchronization Primitives in DML

Learning in Distributed Machine Learning (DML) Systems involves orchestrating and managing numerous worker nodes, particularly in Federated Learning (FL) scenarios with many IoT devices. Synchronization mechanisms on the server side ensure consistent and conflict-free updates to model parameters. Multiprogramming and synchronization primitives are essential for managing access to shared resources and preventing race conditions. This section highlights the principal mechanisms utilized in this context:

Locks Locks are fundamental in distributed machine learning for protecting shared resources, such as model parameters, ensuring that only one node can modify them at a time. Techniques like lock striping and sharding reduce contention by breaking down large locks into smaller, more manageable parts, allowing greater parallelism and reducing performance bottlenecks.

Mutexes (Mutual Exclusions) Mutexes provide a robust mechanism for ensuring exclusive access to critical sections of code, particularly when updating shared model parameters. They prevent multiple nodes from making concurrent updates that could lead to inconsistent states. Priority inheritance protocols help avoid priority inversion by temporarily elevating the priority of the task holding the mutex.

Semaphores Semaphores manage access to limited resources, such as GPUs, in distributed machine learning. By controlling the number of concurrent accesses, they ensure efficient and fair resource utilization. Timeout mechanisms allow tasks to fail gracefully if they cannot acquire a semaphore within a specified time, preventing resource exhaustion.

Condition Variables Condition variables synchronize nodes based on specific conditions or events, allowing threads to wait until certain conditions are met. This is crucial in distributed machine learning for coordinating updates and ensuring nodes do not proceed until necessary conditions are satisfied. Robust signaling mechanisms ensure reliable condition checks and prevent missed signals or spurious wakeups.

Read-Write Locks Read-write locks are particularly useful in scenarios where model parameters are read frequently but updated infrequently. They allow multiple nodes to read data simultaneously while ensuring exclusive access for write operations, improving efficiency and maintaining consistency. Fairness policies help prevent writer starvation by prioritizing writers after a set number of reads.

Ensuring Model Consistency

Ensuring model consistency in DML involves carefully designing synchronization mechanisms to prevent data corruption and ensure coherent updates. Multiprogramming and synchronization primitives are essential for maintaining consistency and efficient resource utilization. By leveraging locks, mutexes, semaphores, condition variables, and read-write locks, server-side systems can effectively synchronize updates, prevent race conditions, and manage access to shared resources. Key considerations include:

1. *Avoiding Deadlocks:* Techniques such as establishing a strict order for acquiring locks and implementing timeout mechanisms can help avoid deadlocks.

2. *Minimizing Lock Contention:* Reducing the granularity of locks and minimizing the duration of critical sections can alleviate lock contention.

3. *Preventing Priority Inversion:* Implementing priority inheritance protocols can mitigate priority inversion by temporarily elevating the priority of the lower-priority node.

4. *Efficient Barrier Synchronization:* Ensuring all nodes reach a certain point in computation before proceeding is critical in distributed training phases, ensuring all nodes have completed their updates before the next iteration begins.

3.5 NS3-FL: Bridging Data & Network Management

It has become clear to this point that one of the key inspirations of this research study is to introduce a user-friendly interface for federating any Machine Learning pipeline in a unified approach that considers diverse datasets, algorithmic implementations and network interaction. Despite the great work being conducted in the field of federated learning by countless researchers, the above venture is something that has yet to be prioritized by an industry leader.

Nonetheless, the paper *NS3-FL: Simulating Federated Learning with ns-3* by Ekaireb et al. [84], introduced an innovative approach that combines a PyTorch-based FL simulator (FLsim) with a network simulator (NS3) to create a unified simulation framework. In this section, we will discuss the architecture and methodology of the proposed framework, which lays the groundwork for integrating our Federated Learning setup with a network simulator.

3.5.1 Framework Overview

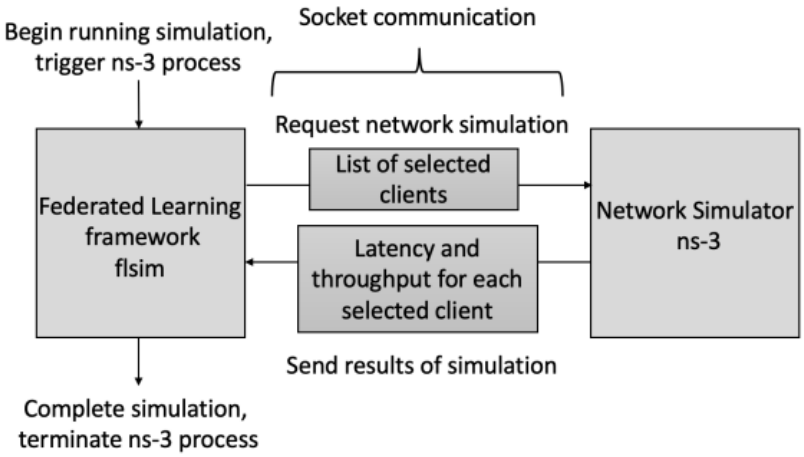


Figure 3.8. *Architectural overview of NS3-FL*

Figure 3.8 depicts the top-level architecture of NS3-FL, which is decomposed into two software blocks in accordance with the previous description: The FL framework of FLsim for data distribution and FL processing, and the NS3 Network simulator for network management and monitoring.

The depicted functionality can be summarized as follows: At the start of each FL training round, FLsim initiates a network simulation request to NS3, specifying the selected clients. NS3 performs the network simulation and returns latency and throughput data for each client. Flsim, in turn, utilizes these network statistics to calculate convergence time and average throughput for the training round.

3.5.2 Learning, Network, and Power Models

Learning Model

The learning model in NS3-FL supports both synchronous and asynchronous FL algorithms. The canonical setting is a star-topology network (see section 2.6.1) with N client nodes connected to a central server. The learning process represents a typical scenario of FL where each client trains on its local dataset and sends model updates to the server. The objective is to minimize

the overall loss function aggregated from all clients' local losses. All the functions regarding the learning process are processed by the FLsim block.

Network Model

The network model in NS3-FL uses NS3 to simulate realistic network conditions. Key metrics such as latency and throughput are considered for each client. These metrics influence the overall convergence time and performance of the FL process. The framework supports diverse network configurations for wireless and ethernet connections.

Power Model

The power model calculates the energy consumption of FL training on client devices by modeling the consumption for two types of appliances: Raspberry Pi (RPi) 4 and 400. In addition, to energy consumption, the time required for training for a specific device is calculated. as the two elements are interconnected $E = P \cdot t$. For the computations, the model accounts for machine-learning-specific parameters such as the number multiply-accumulate (MAC) operations, along with coefficient parameters calculated by applying linear regression on the power measurements for the two devices. This model helps in understanding the performance; including energy efficiency and time for training of different FL configurations.

3.5.3 Implementation

The implementation of NS3-FL involves creating client and server applications within NS3, to mimic the data transmission conducted for the global model distribution and the model updates. The communication between FLsim and NS3 is managed through the interprocess communication protocol that is based on *TCP Sockets*. The framework can be easily extended to include new data, algorithms, and network models, as it is highly configurable through configuration files. Figure 3.9 presents an example encompassing the workflow during a complete training process. The communication between NS3 and Flsim is facilitated by four primary socket commands:

#	Command	Use
0	RESPONSE	Send results from ns-3 simulation
1	STARTSIM	Schedule a round simulation in ns-3
2	EXIT	Terminate ns-3 process
3	ENDSIM	For async, alerts that ns-3 round ended

Table 3.2. *Summary of socket commands used in NS3-FL*

Observing the flow diagram, the FL simulator (FLsim) after starting, initializes the network simulator (NS3). Subsequently, to start a simulation for a training round a request is sent of command type: 1 including the list of participating clients. Then NS3 runs a network simulation round using the selected set of participants. There is a mapping between the two simulators to match their corresponding clients. When the round is complete, a response is sent back to flsim containing the simulated latencies and throughput of the participants. That way the interprocess communication is finalized for this communication round. This process is depicted in fig. 3.9 along with a detailed example.

3.5.4 Simulation Results

The NS3-FL framework was validated through a real deployment using Raspberry Pis as well as large-scale simulations. The validation focused on measuring the framework's accuracy, training

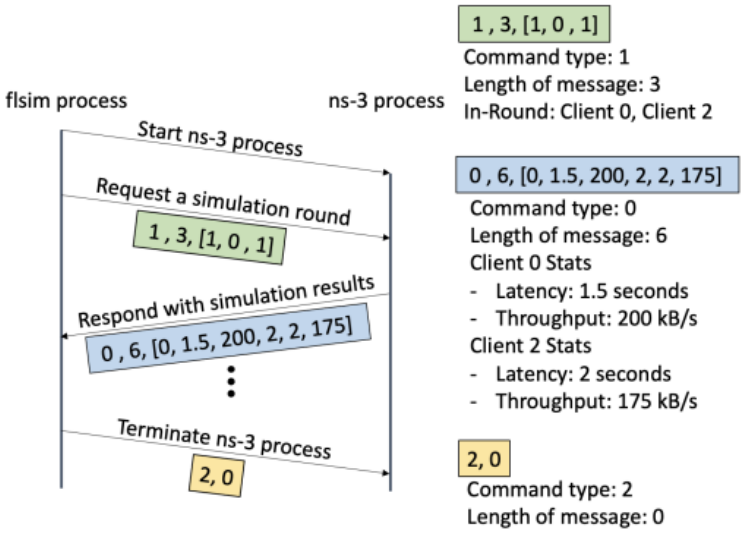


Figure 3.9. *Synchronous Federated Learning training workflow*

time, and energy consumption, comparing these metrics against those obtained from the real deployment. The results demonstrated that NS3-FL accurately replicates real-world federated learning scenarios, with errors not exceeding 1.5%. An illustration of the results is provided in table 3.3 For robustness, the analysis also included varying datasets, local epochs, and client selection strategies.

Experiment	MNIST	FashionMNIST	CIFAR-10
Real, RPi 400	18.32 ± 2.12	29.69 ± 1.10	33.86 ± 6.57
NS3-FL, RPi 400	20.55 ± 1.1E-5	27.78 ± 1.9E-5	30.44 ± 1.4E-5
Real, RPi 4	18.91 ± 0.84	27.85 ± 1.17	30.30 ± 1.47
NS3-FL, RPi 4	19.01 ± 1.1E-5	27.44 ± 1.9E-5	30.15 ± 1.4E-5

Table 3.3. *Comparison of computational time (in seconds) between real deployment and NS3-FL simulations on diverse configurations*

Additionally, a series of simulations were conducted under different data distributions and network conditions to assess the performance and scalability of various federated learning configurations. These simulations provided valuable insights into how different setups affect the overall efficiency and scalability of federated learning systems.

3.5.5 Conclusion

The NS3-FL framework provides a compelling tool for simulating federated learning under realistic network conditions. By integrating flsim with ns-3, researchers can validate their FL algorithms considering the interplay between data, algorithm, and network. The framework's ability to accurately replicate real-world scenarios makes it a valuable resource for advancing federated learning research, as well as, a great starting base for implementing novel approaches in unifying network and data management, similar to the one described in our thesis.

Part II

Technical Implementation

Chapter 4

Environment Replication

In this chapter, we will provide a comprehensive overview of the tools and frameworks utilized for the project's implementation. The integration of various platforms and communication protocols is critical to replicating the environment in which this research was conducted. This includes frameworks for Deep and Federated Learning, network simulation tools, and interprocess communication mechanisms.

To be precise, *TensorFlow* and the *Keras* library were used to develop and train deep learning models, which were then incorporated into the *Flower framework* for Federated Learning (FL). Additionally, *NS3* simulated the network environment to provide realistic conditions for the Federated Learning process. Mechanisms for interprocess communication were essential. In this regard, *POSIX sockets* enabled communication between the network simulation (NS3) and the Federated Learning framework (Flower), ensuring network conditions could dynamically influence the FL results while *gRPC* facilitated efficient communication between the Flower central server and clients, orchestrating the learning process across multiple devices. A brief introduction of the aforementioned tools follows.

4.1 Tensorflow & Keras Deep Learning Platforms

4.1.1 TensorFlow Framework

TensorFlow, developed by Google, is an open-source library for numerical computation and large-scale machine learning (ML). Initially designed to extend ML applicability beyond Google's computational infrastructure, TensorFlow caters to the needs of both inexperienced practitioners and expert researchers. It supports tasks such as deep learning, neural networks, and general numerical computations on CPUs, GPUs, and TPUs. TensorFlow's strength lies in its extensive community, robust scalability, and versatility, enabling it to train and run deep neural networks for tasks like image recognition and natural language processing. It also provides robust tools for model training and optimization, along with infrastructures for efficient dataset handling.

The framework uses a flexible architecture that lets users deploy computation across a variety of platforms (e.g., CPUs, GPUs, and TPUs), including mobile and edge devices. TensorFlow's computational model uses dataflow graphs, where nodes represent mathematical operations and edges represent data arrays (tensors) that flow between them. It also has extended its support in distributed setups with TensorFlow Distribute API which provides the capability of orchestrating multidevice Distributed Learning.

Tensor data arrays are the fundamental building blocks in TensorFlow. They are the generalized version of scalars (0D), vectors (1D), and matrices (2D) and can extend to higher dimensions. Tensors facilitate the representation and manipulation of data in Machine Learning models, allowing for efficient computation across different hardware platforms, including CPUs, GPUs, and

TPUs. Operations on tensors are defined in computational graphs, enabling parallel processing and distributed computing essential for large-scale Machine Learning tasks.

While Python is the primary language for TensorFlow's front-end API, it also supports several other languages, including C++ and Java, facilitating a broad range of applications. TensorFlow's high-level APIs, like Keras, simplify model building and training, while its low-level APIs offer more flexibility for research and experimentation.

Since its release in 2015, TensorFlow has become one of the most popular frameworks in the AI community, used by major companies like Google, Airbnb, Coca-Cola, and eBay. Its ecosystem includes tools like TensorBoard for visualization and TensorFlow Lite for mobile and embedded devices, making it a comprehensive solution for developing and deploying Machine Learning models.

4.1.2 Keras Library

Keras is a high-level, deep-learning API developed by Google for implementing neural networks. It is written in Python and is used to make the implementation of neural networks easy.The Framework is characterized by great adaptability as it can not only be implemented on top of the majority of deep learning frameworks including TensorFlow, PyTorch and Theano, but also used as a tool to export a model's architecture across different platforms. Keras's key characteristics can be summarized as follows:

- *User-Friendly:* Simplifies the process of building and training models, with the standardization of trivial aspects of the process. This way the user's focus shifts towards improved model implementation without distractions. It also contains built-in models for testing and comparisons.
- *Modular and Extensible:* Models are composed of layers, that are easily customizable to fit the user's on-paper requirements without requiring technical know-how.
- *Flexibility:* Provides optimizers, loss functions, and metrics, along with essential functions like "fit" for training and "evaluate" for accuracy assessment. For all the above there are ready-to-use implementations while customization is always available for individual purpose needs.
- *Performance and Scalability:* Widely adopted in industry and research due to its robust performance.

The neural network implementations of keras are based on layer architecture. Layers are the fundamental building blocks used to create models. Each layer performs specific operations on the input data and passes the output to the next layer. Any type of layer located in neural networks in theory can be replicated in Keras including Fully-Connected, Convolutional, Recurrent, Pooling or Dropout layers. For instance, a fully-connected layer can be employed as a *Dense Layer* in a Keras model implementation. That way users can build custom models stacking layers sequentially or using the functional API for more comprehensive architectures.

4.2 Flower Framework: An Efficient Approach to FL

Flower is another open-source framework, supported by a very active community, designed for Federated Learning orchestration. It facilitates the development, experimentation, and deployment of Federated Learning systems, enabling Machine Learning models to be trained across multiple devices or locations while providing efficient communication and privacy handling. In essence, Flower provides a comprehensive API to simplify the numerous complications of Distributed Learning system management by enabling seamless transitioning from ML to FL setups.

This is a vital element of Flower's functionality as it not only simplifies the process but also ensures that it is ML framework agnostic being able to integrate a wide range of ML platforms. This flexibility is achieved through high-level abstraction, allowing users to effortlessly incorporate their learning processes into the client's infrastructure.

Flower comprises the primary tool for our project's implementation offering several advantages for our experimentations. First, it provides the infrastructure both for real-life FL system deployments and simulation settings. This way any given FL system can be tested in optimized simulation setups on a single machine, before proceeding to large-scale real-life deployments. Additionally, with various pre-built aggregation strategies, it simplifies the implementation of Federated Learning models while providing great customizability, letting users tailor solutions to specific implementation needs. Finally, the framework is designed for scalability, making it suitable for large-scale deployments, and it accommodates compute heterogeneity, enabling it to operate efficiently across varying devices and environments.

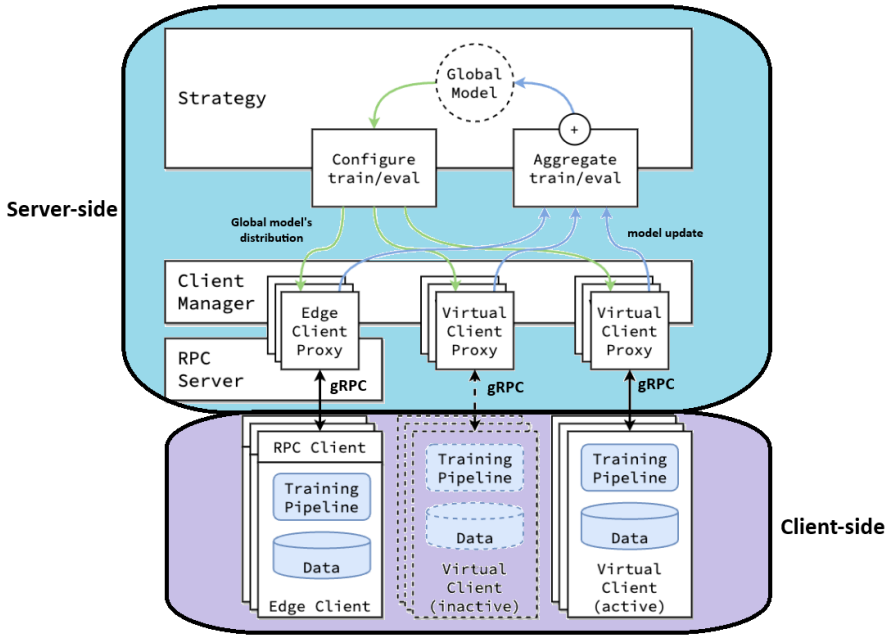


Figure 4.1. *Flower architecture with one server, two edge, and one virtual clients*
Source

Based on fig. 4.1, Flower's architecture can be decomposed into three major parts: the server-side, the client-side, and the interprocess communication protocol.

The server side contains the *global model*, the *strategy*, the *client-manager* that orchestrates the *client-proxies* and an *RPC server*. Starting from the *global model*, it is computed by aggregating the local updates and then redistributed to clients following the FL process. The management of the overall Federated Learning process, including configuring training/evaluation tasks and aggregating results from clients to update the global model is assigned to the *strategy*, which can be a pre-built implementation of a custom logic structured by the user. The *client proxies*, represent an instance of the clients on the server side and act as an intermediary between the RPC server and the edge clients, facilitating communication and data exchange with gRPC calls. The *client-manager* assumes the responsibility of coordinating the interactions between the server and clients. Finally the *RPC server* handles remote procedure calls, enabling communication between

the central server and distributed clients.

The client side of the Flower framework consists of numerous clients, which can be either *virtual* instances or *actual edge devices*. Each client possesses a *local dataset* used for the local training process. In any given training round r, a client may be either *active* or *inactive*. Active clients, whether virtual or edge devices, participate in the training process using their local training pipeline, which can be implemented on their preferred platform. Upon completing training, active clients communicate their results through the *RPC client interface* . Inactive clients remain dormant until they receive a gRPC call from the server, requesting their participation in the subsequent round.

Finally, the interprocess communication is conducted with the employment of the *gRPC protocol*. In a brief summary, the protocol requires an RPC server on the server side and an RPC client on the client side. The RPC Server sends requests and receives responses from the RPC Client on the client side. Active clients use the gRPC protocol to send their local training results back to the server, ensuring efficient and reliable data transmission. This mechanism supports seamless and scalable Federated Learning.

4.3 NS3: An Event-Driven Network Simulator

NS3 is a discrete-event network simulator primarily used for research and educational purposes. What that means is that the simulator models the operation of a network as a sequence of events occurring at specific points in time. Each event represents a distinct change in the state of the system, such as the arrival of a packet, the completion of a transmission, or the movement of a node. The simulator processes these events in chronological order, updating the state of the network accordingly. This approach allows for detailed and accurate simulation of network behavior over time.

Being open-source and highly extensible, NS3 provides a realistic simulation environment for modeling and analyzing the performance of networking protocols and architectures. It supports a wide range of communication topologies, both wired and wireless, with detailed implementations that manage various network layers to closely approximate real-life setups. Notably, in wireless topologies, NS3 can simulate node mobility, interference, and signal propagation.

Additionally, NS3 offers traffic patterns, data flows, and traffic generators to experiment with different scenarios. This capability allows users to create detailed and accurate network architectures, from which they can extract significant performance metrics such as latency, throughput, and jitter. These features make NS3 a powerful tool for investigating and simulating various networking scenarios, providing valuable insights into network performance and behavior.

NS3 benefits from an active community of researchers and developers who continuously strive to enhance the accuracy of its models and the overall functionality of the simulator. It offers comprehensive documentation, tutorials, and example scripts to assist users in getting started. The simulator is designed to be user-friendly, featuring a modular architecture that facilitates easy integration and customization. Moreover, NS3 supports interoperability with real-world systems and other simulators, thereby enhancing its utility for diverse networking research and development projects. These attributes have established NS3 as a leading paradigm in the field, extensively used and taught in universities to foster research and education.

4.4 Posix Sockets & gRPC

4.4.1 Posix Sockets

Posix Sockets are fundamental to network communication, providing a mechanism for two machines to exchange data over a network. Essentially, sockets offer a well-established API for Inter-Process Communication (IPC) in Internet and Unix Domain applications. They present the abstraction of a local endpoint in the communication path, represented as file descriptors. Sockets use the TCP/IP protocol suite for communication and support both connection-oriented (using TCP) and connectionless communication setups (using UDP).

The robust and reliable mechanisms of POSIX sockets make them a viable choice for Federated Learning (FL) inter-process communication. One major property of sockets, making them suitable for FL setups, is their ability to handle both blocking and non-blocking communication enabling them to support synchronous and asynchronous FL. In a blocking operation, the program halts until the entire message is sent or received, which can cause potential delays. Conversely, in a non-blocking operation, the program sends or retrieves only the data that is immediately available, preventing it from stalling on slow connections and avoiding many deadlock situations.

In our case, socket communication is restricted to operations within the NS3 simulator, specifically handling the internal processes and simulations. This includes managing the simulated network events and data exchanges. Once the simulation tasks are completed, sockets are used to communicate the results from NS3 to the Flower framework. This ensures that the network simulation outcomes are efficiently transferred for further processing in the Federated Learning workflow managed by Flower, enabling seamless integration and coordination between network simulations and Machine Learning tasks.

4.4.2 gRPC

gRPC is an open-source RPC (Remote Procedure Calls) framework developed by Google, designed for efficient, low-latency communication between distributed systems. It allows the server to call methods on clients as if they were local, using Protocol Buffers (protobufs) for serializing structured data, ensuring fast and compact communication. gRPC supports multiple programming languages and offers features such as authentication, load balancing, and bidirectional streaming, making it ideal for microservice architectures where seamless and scalable service communication is essential. Its high performance and ease of integration make gRPC a popular choice for modern distributed computing environments.

The process of RPC communication with gRPC begins by defining a service in a *.proto* file using Protocol Buffers' syntax, where service methods and message types are specified. This definition is compiled using the `protoc` compiler to generate both client and server code. The server implements the service by providing the logic for handling RPCs, processing requests, and sending responses. Meanwhile, the client creates a stub from the generated code, enabling it to invoke remote methods on the server as if they were local functions.

It is important to clarify that in this context, the terms *Server* and *Client* are used in a general sense for communication purposes, not specifically for Federated Learning structures. The reason for this comment will be better demonstrated with the following example.

In our project the server orchestrates training and evaluation, maintaining client stubs within the client proxies. This setup allows the server to remotely call the clients' fit and evaluate methods to initiate the learning process. Once the results are generated, they are returned to the client proxies and subsequently to the server. Here, the client proxies, and indirectly the Flower server, function as the gRPC client, while the Flower clients act as the gRPC server.

Chapter 5

Methodology

This chapter provides a holistic overview of the methodology employed in this study, enabling readers to understand and reconstruct the proposed architectuovers the top-level architecture, component analysis, dataset management, model selection, and the integration of these elements, concluding with a summary of the performance metrics used for the system's evaluation.

To accurately demonstrate the proposed approach, we revisit the primary objectives of this research, highlighting the connection between the goals and methodology. This thesis aimed to develop a comprehensive framework for Federated Learning that enables the seamless implementation of Federated Learning systems while emulating realistic network conditions. The communication component requires a simulator for accurate time measurements because this study also employs an advanced synchronization method inspired by the Geometric Method (GM)[17], known as FDA, which significantly mitigates communication overhead. To measure the efficiency and real-life impact of this implementation through the University's lab infrastructure, a realistic simulation environment that can replicate actual communication conditions was required. The following sections will delve into the methods and techniques used to construct the necessary infrastructure and synchronization logic for this study.

5.1 System Architecture

The top-level architecture of fig. 5.1 demonstrates the cooperation between the Federated Learning and Network Simulation components. Elucidating this, on the left side, we have the Federated Learning Orchestrator represented by the Flower Framework. Flower serves as the cornerstone of the implementation, providing the entire infrastructure that enables the Federated Learning process. The server, client, and synchronization logic are all based on the simple and customizable interface provided by this open-source framework. As established earlier, the framework supports a wide range of deep learning backends. For this project, we have chosen TensorFlow, enhanced with the additional capabilities provided by Keras.

On the right side, we have the Network Simulator component implemented by NS3. This component is responsible for emulating realistic network conditions, ensuring that the communication aspect of the Federated Learning process is accurately represented and measured.

Upon starting the application, the Federated Learning server initiates the Network Simulator and sets it into operation. Concurrently, the server awaits client connections, with each client registering one by one on the server's client manager. For each registered client, a proxy is created on the server (see fig. 4.1), enabling the management of remote calls to the clients. These proxies are assigned unique arithmetic IDs, shared between Flower and NS3, to establish connections between the Network Simulator and the Federated Learning clients. Once the requisite number

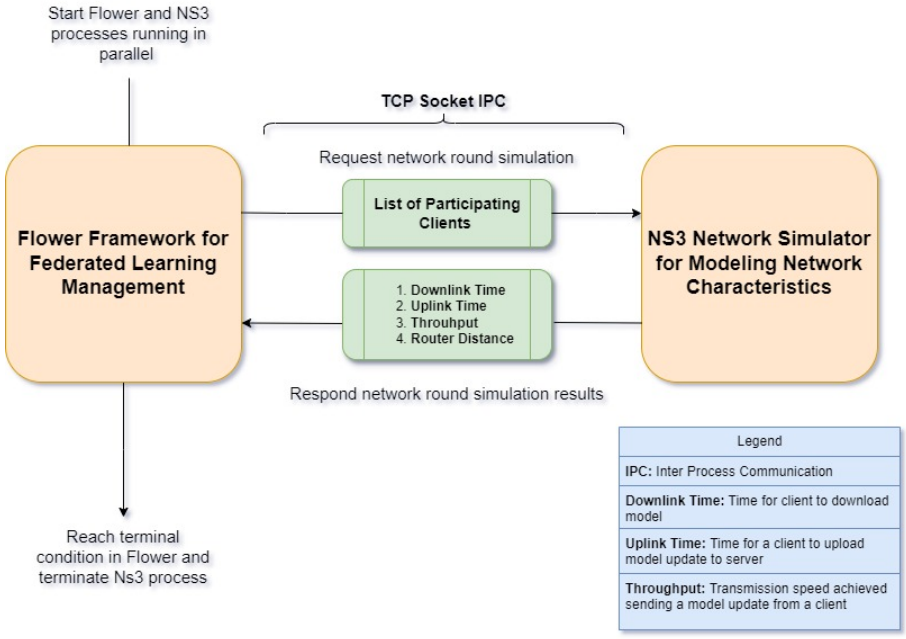


Figure 5.1. *Top-level architecture combining Flower Framework for Federated Learning with NS3 for network simulation*

of clients is connected[1], the server commences the Federated Learning procedure.

At this point, the client selection phase is conducted, as described in section 2.5.4, marking the first step of the learning procedure. Utilizing the mapping formed during registration, the selected set of client proxies can be mapped directly to the unified ID used by both sides. To avoid confusion, consider the following example: client proxies with IDs *0, 2, 3* are selected to participate in a training round, with the total number of clients connected to the server being *5*. Along with the request for a simulation round to NS3, the Flower Server will additionally pack an array of size equal to the total number of clients, containing binary values that signal their participation status. Thus, the following list represents the participating clients.

$$\text{participants: } \{0,2,3\} \implies [1,0,1,1,0]$$

The Network Simulator decrypts the round initiation message as explained in section 3.5 and commences a simulation round where only the specified clients participate. A simulation round involves both setting up the network and running the transmission simulation. Further information about the infrastructure will be provided in the forthcoming analysis, as the current requirement is to establish a clear understanding of the top-level architecture.

The metrics of interest are stored in a data structure that maps each client's results to their corresponding ID. The key metrics monitored as drawn from fig. 5.1 are as follows:

- *Downlink Time:* This metric measures the time it takes for a client to download the global model distributed by the server at the start of a Federated Learning (FL) round. The time required to establish the connection is also considered.

- *Uplink Time:* This metric records the time required for a client to upload their model update to the server.

[1]The minimum number of clients required for an operational setup is specified as an argument to the server.

- *Throughput:* This metric captures the throughput during the uplink procedure, reflecting the speed of the client's data transmission.
- *Router Distance:* This metric measures the distance between the client node and the router. A greater distance from the router typically results in lower signal power, leading to slower transmission rates, or even worse dropouts in the case that the router cannot be reached.

Additionally, we monitor an indirectly induced value: whether a client drops out due to network-related issues. Specifically, this occurs if there is a failure to either receive the global model or send the model update. This metric is critical because if a client fails during the communication simulation, they are unregistered from participating in the subsequent Federated Learning round on Flower. This process emulates realistic failure conditions.

Following the calculation of communication metrics, the results are returned to the Flower module. As stated above, dropouts are used to determine which of the clients spawned by the client selection process will ultimately participate in the training round. The remaining metrics are cataloged, and the federated learning procedure takes place.

The Inter-Process Communication (IPC) that unifies the Federated Learning module with the Network Simulator module is managed using the Unix Socket communication protocol. Specifically, the structure of this communication mirrors that utilized in *NS3-FL* as described in section 3.5. An interface for handling inter-process communication is implemented on both sides, featuring a restricted range of communication commands. These commands include *StartRound*, *Response*, and *Exit*.

This cycle is repeated until the desired number of rounds is executed. The definition of a *round* can vary depending on the setup or literature approach. For this research, the definition used is the one employed by the Flower Framework infrastructure, which represents the number of model update communications. To improve the computation-to-communication ratio, a round can often contain more than one epoch, as discussed in section 3.4. This is particularly true for batch learning settings (see section 2.1.2). However, in online learning situations, where new data is continuously generated, the traditional definition of an epoch as a pass over the entire dataset cannot be applied. Considering this, in this implementation, the terms *round* and *epoch* will be used interchangeably.

5.2 Dataset Preprocessing & Management

In an effort to highlight the immense potential of FDA in various scenarios, we tested our implementation over the benchmark datasets discussed in section 3.1.1, including *MNIST*, *FashionMNIST*, and *CIFAR10*. These tests were conducted on both homogenous and heterogenous datasets to evaluate the performance and robustness of the FDA approach under different data distributions.

However, dataset selection is not the primary focus of this section. Instead, the emphasis here is shifted toward the dataset preparation techniques employed to preprocess our data, ensuring they meet the requirements for our Online Distributed Learning framework.

Our implementation adheres to the core principle of Federated Learning (FL), ensuring that the server retains no knowledge of the clients' local data. Unlike traditional Distributed Machine Learning (DML) approaches, where the dataset is prepared on the server and then distributed, our method has each client retrieve its data partition directly. Each client uses a `get_dataset` function with a unique partition ID to obtain its dataset segment. The dataset is divided into *numClients* partitions, with client_0 retrieving partition 0, client_1 retrieving partition 1, and so on. This approach preserves data privacy and adheres to the FL principle of local data ownership.

At this point, the dataset preparation process will be presented, initially for the scenario of i.i.d data and subsequently taking into account data heterogeneity. The steps for preparing an iid dataset for a client are as follows:

1. *Load Data:* The datasets are derived from Keras. This ensures the data are pre-split into training and test sets, simplifying the initial data handling process.

2. *Data Preprocessing:* This step involves reshaping the data into the format required by the CNN: `(batch_size, height, width, channels)`. This is necessary because datasets are often stored as one-dimensional arrays that need to be reshaped for proper processing. Following reshaping, the data are normalized to ensure stability and efficiency during training.

3. *Data Shuffling:* The data are shuffled to improve generalization by creating random samples. This step helps in reducing bias and ensuring that each partition is representative of the entire dataset.

4. *Data Partitioning:* The data are divided into *numClients* partitions. Each client is assigned the partition corresponding to their partition ID, ensuring an even distribution of data across clients.

5. *Data Train-Test Split:* Within each client partition, the data are further split into training and test sets. If necessary, the initial client partition can also be split into an additional validation set for hyperparameter tuning. For example, if the initial client partition contains 10,000 samples and the ratios for test and validation are set to 0.1, the resulting training set will contain 8,000 samples, while the test and validation sets will each contain 1,000 samples.

6. *Trainset Repetition:* To simulate an online dataset, the training data undergoes three transformations: it is shuffled, cached, and finally repeated. These transformations help in mimicking an online-generated dataset, ensuring continuous availability of training data.

In the case of non-iid data, the steps for shuffling and partitioning (steps 3 and 4) are adapted to generate biased partitions. To achieve the desired non-iid output, we have created four distinct templates of bias: random bias, small bias, medium bias, and large bias. These templates allow us to systematically control the extent of heterogenous characteristics present in the data, thereby better simulating real-world scenarios. To ensure repeatability and maintain different distributions for each client, the client ID is used as a seed to produce consistent yet varying results across experiments.

For small bias, we utilize the normal distribution to generate partitions, ensuring a slight imbalance in the data distribution across clients. This method introduces minor variations while maintaining overall data diversity. For greater bias, we employ the widely applied Dirichlet distribution, which is an industry-standard for generating non-iid distributions. The Dirichlet distribution allows for flexible control over the level of heterogeneity among client datasets. By adjusting the concentration parameter (a), we can control the degree of bias: lower values of a result in higher bias, while higher values lead to more balanced partitions[85].

5.3 Model Development and Training

The availability of a plethora of datasets necessitated the use of various Artificial Neural Networks (ANNs), or more simply, models. The simulator accommodates a wide range of models with varying complexities to uncover the underlying patterns within the datasets. The following table provides an introductory overview to illustrate the connections between datasets and their corresponding models.

Dataset	LeNet-5	VGG16	ResNet50	Default
MNIST	✓	×	×	✓
FashionMNIST	✓	×	×	✓
CIFAR10	✓	✓	✓	✓
CIFAR100	✓	✓	✓	✓

Table 5.1. *Supported Datasets and ANNs*

Not all model and dataset combinations are displayed in this thesis dissertation, but experimentations can be performed without any changes to the models. Details about the considerations for the presented experimental parameters will be further discussed in the following chapter.

As observed in the table above, besides the widely used ANN architectures of *LeNet-5*, *VGG16*, and *ResNet50*, it is important to highlight the "Default" column. If no specific ANN is specified during experimentations, the simulator is equipped with a series of default, simple model implementations. These default models are designed to provide adequate learning capacity while maintaining a restrained size, thereby preventing excessive burden on the network due to large model transmissions. There is one Default implementation provided for each dataset, each with scaling capabilities to match the corresponding model's complexity.

5.3.1 Default Model Architectures

The layered architecture of the default models is displayed in appendix A. Nonetheless, a brief description of their architecture is provided here to clarify their operation before proceeding with the experiments that utilize them.

MNIST Model This model is designed for the MNIST dataset, which consists of grayscale images of handwritten digits. The model is relatively simple, inspired by the LeNet-5 architecture. It includes a single convolutional layer followed by a max pooling layer, which helps in reducing the spatial dimensions and extracting features. After flattening the output, a dense layer with ReLU activation is used for further processing, and the final dense layer with softmax activation is used for classification into 10-digit classes. It is comprised of 347,146 parameters.

FashionMNIST Model The model for Fashion MNIST is more complex, reflecting the slightly more challenging nature of the dataset, which contains grayscale images of various clothing items. This model is inspired by more advanced architectures that incorporate batch normalization and dropout for regularization. It consists of two convolutional layers with max pooling and dropout, followed by another dense layer with ReLU activation and batch normalization to enhance learning. The model aims to capture intricate patterns in the fashion items, improving generalization and performance. It is comprised of 898,314 parameters.

Cifar10 Model This model is designed for the CIFAR-10 dataset, which consists of small color images in 10 classes. The architecture is inspired by the VGG network, known for its simplicity and depth. It includes five convolutional blocks, each with multiple convolutional layers followed by batch normalization and dropout for regularization. Max pooling layers are used after each block to reduce the spatial dimensions. The model ends with fully connected layers to perform the final classification. This deep architecture allows for capturing complex features necessary for accurate image classification in CIFAR-10. It is comprised of 15,262,026 parameters.

The above model implementations are not optimized and fine-tuned for maximum performance and are profoundly inferior to the state-of-the-art models commonly used in the market. However, their primary characteristic that makes them a viable choice for certain experiments is their

compact size, which allows them to deliver respectable results without excessively burdening the computational and communication resources.

5.3.2 Custom Training Logic for FDA

The implementation of the geometric method and FDA for synchronization demands custom communication adaptation. As thoroughly studied in section 3.4.2, this technique requires custom logic for per-batch synchronization. To achieve this, we engineered a custom model extending Keras functionality, which behaves as desired and keeps track of specific parameters. Overall, the custom training and evaluation logic in the `CustomModel` class is designed to provide fine-grained control over the training process, incorporating custom metrics and callbacks to facilitate distributed training and model tracking.

The custom training technique will be described briefly. The training dataset is batched according to the specified batch size. A loop iterates over these batches, performing a forward pass and calculating the loss for each batch. Gradients are computed using `tf.GradientTape` and applied to the model's trainable variables using the ADAM optimizer. Metrics such as accuracy, mean loss, and L2 norm are updated during each batch iteration. The L2 norm is a custom metric that tracks changes in model weights, calculated as the difference between the current model parameters and the parameters received at the start of the round, as discussed in section 3.4.2.

Following the Keras model's training approach, callback function support is provided at multiple points within the custom training loop. This allows users to execute specific operations at different stages of the training without direct access to the training logic. A focal point of our implementation is the `on_train_batch_end` callback function, which is called at the end of each batch to trigger the communication of the L2 norm—a float value sent to the metric server for tracking.

To clarify the synchronization in the Naive FDA approach, the logic dictates that after each processed batch, the L2 norm calculated between the initial weights W_{t0} and the current weights W_t of the model is communicated to a metric server. This server, distinct from the Flower server orchestrating the Federated Learning process, solely monitors the L2 norm. Upon receiving the expected number of L2 norms [2], the average L2 norm is calculated and compared against the RTC threshold Θ. If the threshold is not exceeded, the training process continues. However, if the threshold is breached, the metric server signals the clients to stop training and aggregate their results. The aforementioned process is displayed in figures 5.2 and 5.3.

5.4 Federated Learning Implementation

5.4.1 Synchronization Approaches

To ultimately demonstrate the effectiveness of FDA, two discrete Synchronization methods for FL are provided: the naive FDA approach described above, and an adjusted implementation of Mini-Batch Synchronous SGD for online data processing, referred to as Online Synchronous SGD (OSyncSGD) (detailed in section 2.4.2). These implementations are specifically tailored for online learning scenarios akin to those investigated in this research. An illustration of FDA's functionality is provided in figures 5.2 and 5.3 for better understanding:

The OSyncSGD synchronization algorithm is a straightforward implementation used as a benchmark for comparison with FDA. As there is no concept Essentially, OSyncSGD is a direct application of Stochastic Gradient Descent, or Mini-Batch SGD to be exact, in a distributed setup. In this method, one randomly selected mini-batch from the training dataset is used to

[2] The anticipated number of L2 norms is equal to the number of clients participating in the training process.

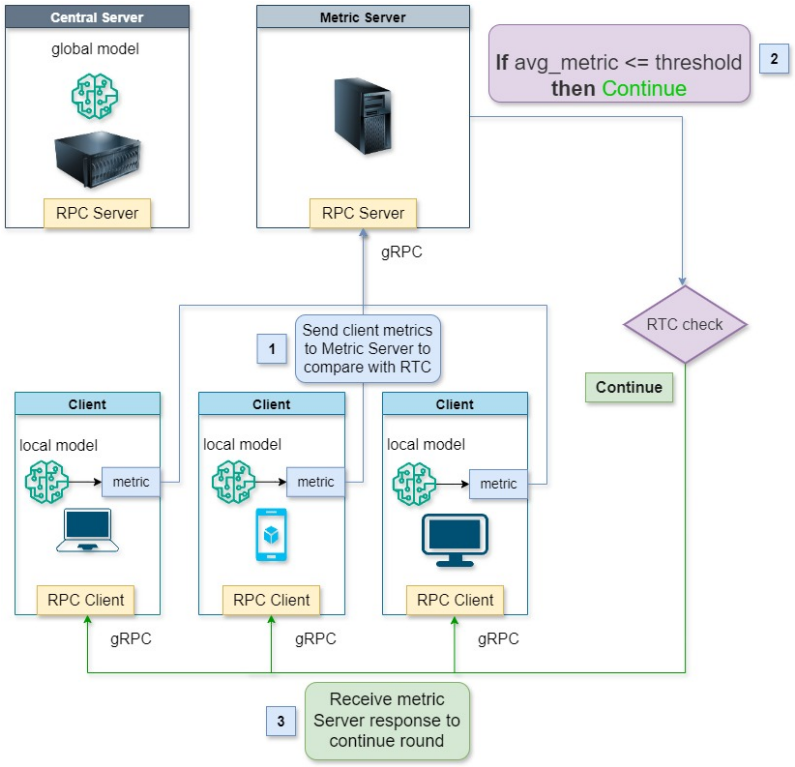


Figure 5.2. *FDA Round meeting the RTC threshold*

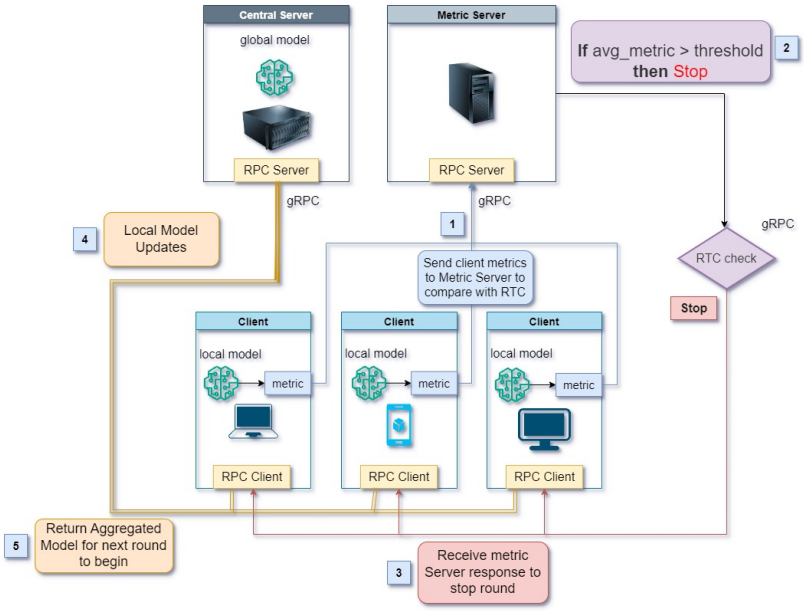


Figure 5.3. *FDA Round breaching the RTC threshold*

calculate the updated local model via gradient descent. This model update is then communicated to the server for aggregation to update the global model parameters, securing per-batch model synchronization. By following this procedure, OSyncSGD provides a baseline approach for evaluating the performance improvements achieved by FDA. The simplicity of OSyncSGD allows it to serve as an effective benchmark for onlines settings, demonstrating the potential benefits of more sophisticated synchronization techniques like FDA.

The Naive FDA approach can be simplified to an OSyncSGD implementation by setting the

threshold Θ to zero. In this configuration, model aggregation is conducted with each batch, with the only difference being redundant communication with the metric server, which always stops the round. This adjustment ensures model aggregation happens in every iteration, mimicking OSyncSGD. In our experiments, OSyncSGD is applied by modifying the FDA implementation to set the threshold to zero and disregard the RTC monitoring process.

5.4.2 Employement of gRPC Communication Protocol

Up to this point, interprocess communication between server and client for RTC monitoring has been analyzed, nevertheless, the protocol utilized for communication has yet to be clarified. The selection of the IPC protocol was inspired by the Flower optimized communication approach that employs gRPC communication. As already discussed in section 4.4.2 gRPC is an extremely efficient IPC commonly utilized in Distributed System scenarios, making it an ideal choice for managing the custom-implemented communication between the aforementioned entities

The performance of gRPC can be attributed mainly to the exploitation of protobuffers for serialized communication. The communication protocol is defined in a `.proto` file, specifying the services and messages used for interactions between the metric server and clients. The `.proto` file serves as the schema for the structured data exchanged, ensuring consistency across components. Using the Protocol Buffers compiler (`protoc`), the `.proto` file is compiled to generate Python classes that handle message serialization and deserialization. These classes abstract the serialization logic, allowing developers to focus on application logic. The compilation results are:

- `metric_service_pb2.py`: Contains classes for the defined messages.
- `metric_service_pb2_grpc.py`: Contains classes for the gRPC services.

Implementation of the Metric Server The metric server implements the gRPC service defined in the `.proto` file. It processes incoming requests, performs operations, and returns responses. It uses a thread pool to handle concurrent client requests, ensuring scalability and responsiveness. The main request processed by the server is the `SendMetricAndWait`. The request contains the L2 norm of the corresponding client as a message and the server returns a boolean response signaling whether to continue training or not.

Implementation of the Metric Client The metric client sends requests to the metric server and processes responses. Clients collect metrics like ahe L2 norm during model training. These metrics are encapsulated in protobuf messages and sent to the server via gRPC for processing.

Communication Workflow

1. *Initialization*: The metric server starts and listens for gRPC requests on a specified port.
2. *Request*: *Client-side*: Constructs and sends request messages using protobuf classes. These messages contain the L2 norm computed between the current model weights and the initial weights (w_{t0}) at the start of the round.
3. *Response*:
   - *Server-side*: Receives and processes requests, and constructs response messages. The L2 norms sent by the clients are received and assigned to worker threads for processing. These threads are paused, applying conditional variables logic (see Section 3.4.5), until all client norms are gathered. Once this happens, the RTC condition is checked (see Figure 5.2) and a

broadcast signal is transmitted to resume the threads, creating a response that determines whether the clients should continue or stop the training round.

- *Client-side*: Receives and processes response messages that command whether the round should continue.

This workflow ensures efficient communication of metric data, facilitating real-time tracking and decision-making. It can be further investigated in fig. 5.5 for better understanding.

5.5 Network Simulator Integration

5.5.1 NS3 Program Component Analysis

The second fundamental module in our thesis implementation is the NS3 Network Simulator. This component recreates realistic communication conditions, reflecting real-world scenarios. It provides a clear understanding of the performance of FL systems beyond the near-ideal laboratory network environment. Federated Learning is commonly used in scenarios with varying network capabilities, as discussed in section 2.6.2, making it crucial to assess the learning procedure's performance under such conditions.

The integration of the Network Simulator is illustrated in fig. 5.1. The top-level architecture shows that the network simulator logic is modularized outside the FL framework, maintaining a clear delineation of responsibility. This ensures the network simulator can independently recreate diverse conditions, crucial for evaluating the robustness and efficiency of Federated Learning systems under varying communication constraints.

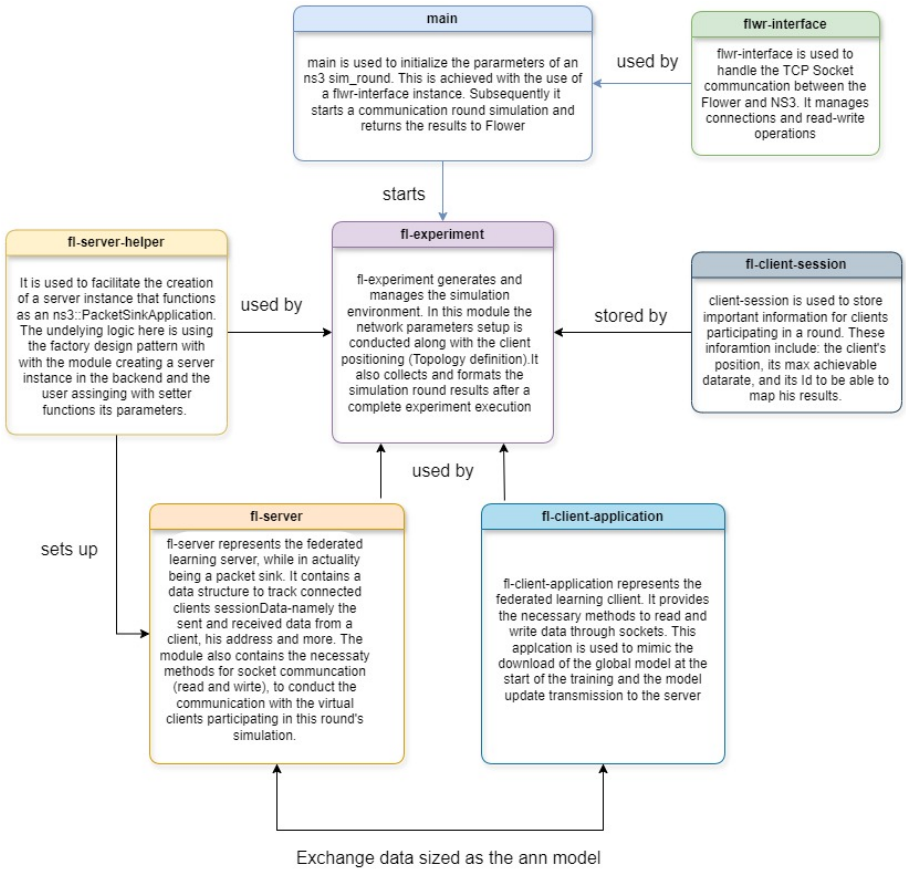


Figure 5.4. *NS3's architecture decomposition*

The architecture depicted above illustrates the complete implementation of a Federated Learning Network Simulator designed in C++ within the NS3 simulator environment. This setup is appropriate for assessing how different network conditions impact the performance of Federated Learning. This approach provides valuable insights into the scalability and adaptability of Federated Learning algorithms in real-world applications.

The process is initiated by the `main` function, which sets up the parameters for the NS3 simulation round received via the `flwr-interface`. This interface manages the TCP socket communication between Flower and NS3. The `fl-experiment` module generates and manages the simulation environment, configuring network parameters and client positioning, and collects and formats simulation results. The `fl-server-helper` facilitates the creation of a server instance acting as an NS3 `PacketSinkApplication` using a factory design pattern. The `fl-server` represents the federated learning server, managing communication with virtual clients by tracking session data. The `fl-client-session` stores crucial client information, including positions, and IDs. The `fl-client-application` represents the federated learning client, providing methods to read and write data through sockets, simulating the download of the global model at the start of training and the transmission of model updates, thus exchanging data sized as the ANN model.

5.5.2 Holistic Workflow Post-Integration

A holistic representation of how the entirety of placeholders in the simulation process cooperate to produce the results of one round is presented in the following sequence diagram:

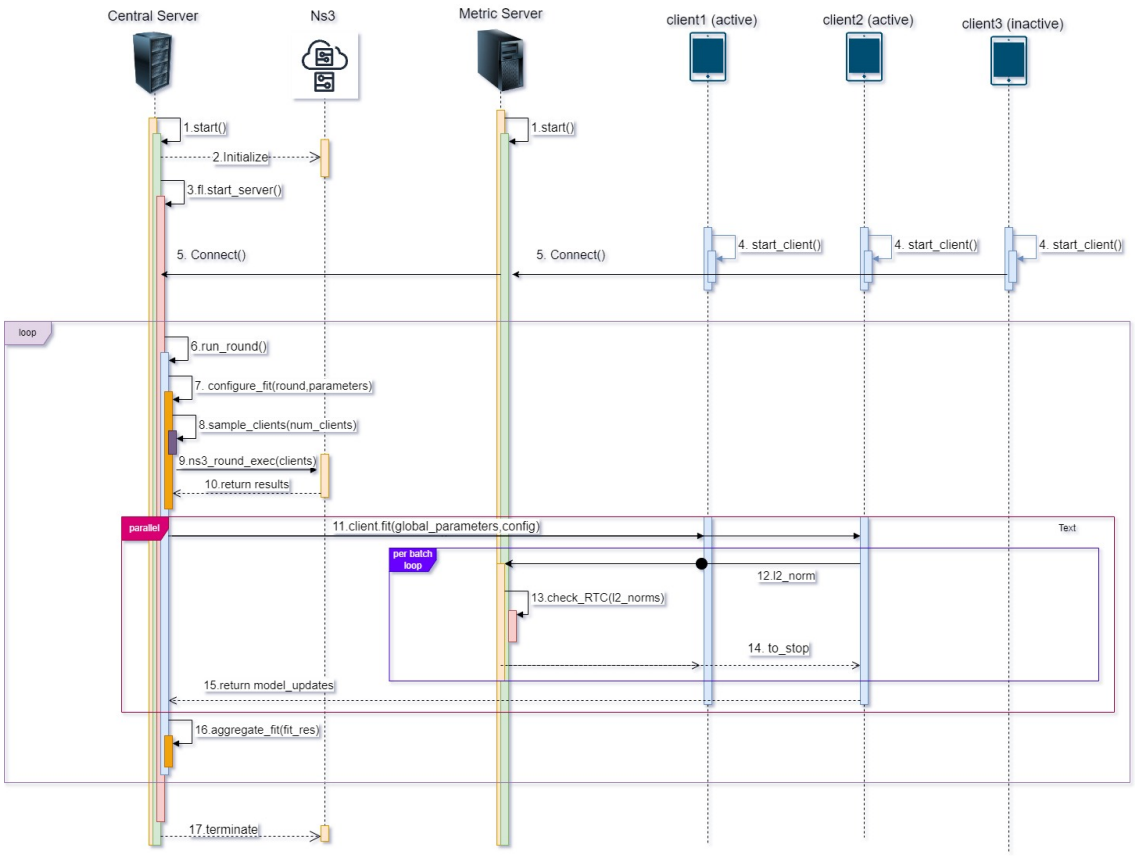


Figure 5.5. *Implemented workflow depicted in a holistic sequence diagram*

The sequence begins with the initialization phase, where the primary orchestrating components of the system are started. The Central and Metric Servers are commenced by invoking the

`start()` method on each. Subsequently, the Central Server instructs the NS3 Network Simulator to initialize its environment asynchronously. Finally, the desired number of clients is initialized with `start()` after the servers and NS3 have been prepared, and they connect to the servers. The active reference in the client names indicates whether a client participates in the current training round.

Once the initialization is complete, including the client and server setup, the system enters the round execution phase, which involves multiple steps. The Central Server enters a loop for a predefined number of communication rounds. A new training round commences with the `run_round()` function. The Central Server configures the training round parameters through `configure_fit(round_parameters)`. During the execution of this function, a set of clients to participate in the current round is sampled with `sample_clients()`.

The sampled set of clients is then passed as an argument for the execution of one simulation round in NS3 via the `ns3_round_exec(clients)` call. Notably, this call in the sequence diagram is performed in a synchronous manner, meaning that the Federated Learning execution blocks while waiting for the results of the NS3 round. The reason for this is that the network simulator results may indicate that a client failed to communicate, signaling their non-participation in the following round. In this case, the client is characterized as a "dropout" and is removed from the set of participating clients.

After the participating clients receive their instructions for the round through the `configure_fit(round_parameters)` execution, the round execution proceeds. During the round execution, client training is conducted in parallel. The `client.fit(global_parameters, config)` method is called for each active client.

Within the client training loop, each batch triggers the calculation of the L2 norm, which is then communicated to the Metric Server. The Metric Server utilizes conditional variables to pause execution for the client threads that await to be resumed when the desired number of l2 norms is received. When the L2 norms of all clients have been collected, the Metric Server checks the RTC (Round Termination Condition) based on the aggregated L2 norm value. If the threshold is breached, the Metric Server instructs the clients to stop training and return their results. The model updates are then returned to the Central Server, which aggregates the model updates and finalizes the round in preparation for the next iteration.

After the pre-declared number of training rounds is reached, the Central Server terminates NS3 execution, closing the communicating sockets. Eventually, both the Central Server and Metric Server terminate their execution.

5.6 Evaluation Metrics and Performance Analysis

5.6.1 Model Evaluation Metrics

In evaluating the performance of the models, several key metrics are employed to provide a comprehensive understanding of their effectiveness. The primary metrics used include the *L2 norm*, *mean loss*, and *accuracy* achieved. The *L2 norm* is calculated to track changes in model weights, serving as a custom metric to measure the difference between the *current model* (W_t) parameters and those at the *start of the round*(W_{t0}). *Mean loss* is another critical metric, reflecting the *average loss* incurred across all batches processed in the round, thus indicating the model's overall performance during training. Lastly, *accuracy* is measured to determine the proportion of correctly classified instances out of the total instances, providing a direct measure of the model's predictive capability.

5.6.2 Network Simulation Metrics

The performance of the network simulation is assessed using several key metrics. *Downlink time* is a crucial metric that measures the time taken for a client to receive the global model from the server, including the setup time. *Uplink time* measures the duration required for a client to upload its model update to the server. *Throughput* is monitored to determine the speed of data transmission during the uplink process. Another significant metric is the *distance from the router*, as the position of clients may vary. The distance is vital because clients located further from the router are more likely to experience slower network speeds and higher dropout rates due to weaker signal strength. Finally, one last monitored metric is the *dropout rate* exhibiting the stability of the network. Those are the directly measured metrics for clients.

In addition to them, aggregated metrics are also recorded regarding the overall system's time measurements, including the average time spent in uplink, downlink and RTC monitoring as well as the total times spent in computation, communication and ultimately in the round.

5.6.3 Performance Analysis

Performance analysis is conducted by examining metrics relative to the *convergence* as well as the *comminication efficiency*. To be precise, regarding the *convergence*, factors that are measured encompass the *maximal accuracy* achieved in same amount of steps, the *speed of convergence* meaning the time it took to achieve a certain accuracy target and the *stability* of the learning curve. On the other hand, when referring to the *communication efficiency*, a key aspect is the computation-to-communication ratio which highlights the proportion of time spent on computation relative to the communication time. Also the decomposition of communication time is studied to locate potential bottlenecks in the communcation.

5.6.4 Metrics Processing

For logging, processing, and visualizing the experimental results, the Python `pandas` library was utilized. Each experiment was assigned a unique identifier (ID) corresponding to the date and time of the experiment's execution. This ensures precise tracking and reproducibility of results. Additionally, a file named `info.csv` is placed in the experiment folder, containing all the parameters of the experiment.

To organize the data efficiently, separate directories for the *server* and *clients* were created. In the server directory, the file `server.csv` contains aggregated results about the federated learning process. Conversely, within the *clients* directory, individual folders were created for the clients selected for bookkeeping, as cataloging data for all clients would introduce significant overhead. Each client's folder contains two files: `epoch.csv`, which records epoch-level data, and `batch.csv`, which records training step data.

Data management is performed using pandas DataFrames, with a distinct DataFrame for managing each of the aforementioned files. This approach allows for seamless integration of data management with the federated learning process, leveraging the extensive collection of tools provided by pandas for data cataloging and manipulation. Consequently, this method facilitates efficient data handling, enabling comprehensive analysis and visualization of experimental results.

Chapter 6

Experimental Results & Discussion

6.1 Overview of Experiments

This chapter presents the experiments conducted to assess the implemented system, along with a thorough commentary on the research findings regarding our initial objectives. Initially, we aim to assess the efficacy of the Naive Functional Dynamic Averaging (FDA) synchronization technique, and by extension the Geometric Method (GM), compared to the previously described Online Synchronous SGD (OSyncSGD) approach. Subsequently, we evaluate the performance of the Federated Learning (FL) framework integrated with the NS3 network simulator, with the primary pursuits of demonstrating the system's scalability, robustness, and adaptability. Keeping these objectives in mind, a series of experiments were conducted covering various scenarios to provide a comprehensive evaluation of the proposed methodologies. An outline of the experiment results to be displayed is presented next, highlighting the relevance to our objectives:

1. *Comparison of FDA vs. Online SyncSGD Approach:*
   - **Description:** Each synchronization algorithm is tested over the same number of training steps in four different client population scenarios. Specifically, experiments are executed for both algorithms with 5, 10, 15, and 20 clients participating in training.
   - **Relevance:** This comparison aims to demonstrate the efficacy of the FDA approach compared to a simplistic alternative and to analyze its scalability.
2. *Impact of Varying Thresholds:*
   - **Description:** A baseline experiment configuration is analyzed for four distinct threshold hyperparameter (Θ) values in the FDA approach.
   - **Relevance:** This analysis examines the effect of hyperparameter tuning on performance, providing insights into optimal threshold selection.
3. *Diversity of Datasets and ANN Architectures:*
   - **Description:** The baseline approach is applied to the FashionMNIST dataset using the default ANN architecture designed for this dataset (appendix A). To generalize the results, additional experiments are conducted with FashionMNIST using LeNet-5, MNIST using both LeNet-5 and the default architecture, CIFAR-10 with the default architecture, and CIFAR-100 with ResNet-50.
   - **Relevance:** This set of experiments evaluates the adaptability of the FDA approach and the overall system across diverse datasets and ANN structures, providing performance benchmarks for the tested combinations.

4. *Handling Non-IID Data Distributions:*

   - **Description:** Experiments are conducted for each of the three bias templates of non-IID data (small, medium, and large heterogeneity) in scenarios with a restricted number of clients (5) and a larger client pool (15).
   - **Relevance:** These experiments analyze the robustness of the system in non-IID data distributions and its adaptability to adverse learning scenarios.

5. *Diverse Network Condition Scenarios:*

   - **Description:** The system is tested under different network conditions using three network condition templates: weak older WiFi, mediocre 2.4GHz WiFi connection, and fast and stable WiFi 6 connection. Additionally, scenarios with both stationary and mobile clients are investigated.
   - **Relevance:** This analysis assesses the robustness of the system under various network conditions and examines the quality of learning in adverse conditions.

To maintain clarity and consistency, the results of the aforementioned experiments will be presented in self-contained sections. Each section will include detailed measurements organized in tables and figures, accompanied by an in-depth discussion analyzing the findings. This structured approach ensures a seamless understanding of the experimental outcomes while reducing the need for constant referencing. Consequently, it establishes a coherent flow of analytical thought, enhancing the overall readability and comprehension of the thesis.

6.2 Experimental Infrastracture

The infrastructure provided by a local machine could not possibly handle the immense load of simultaneous, real-time testing of advanced ANN architectures and diverse client pool populations[1]. Consequently, the experiments were conducted on the Polytechnix server station in the laboratory of the Technical University of Crete (TUC) via SSH, able to support parallel data processing for multiple clients. The server station featured high-performance computational resources, including multiple CPUs, substantial RAM, and high-speed storage, ensuring that the experiments were conducted under the required conditions. The hardware sheet of the utilized Distributed System is as follows:

Component	Description
CPU	Intel(R) Xeon(R) Silver 4310 CPU @ 2.10GHz 48 CPUs, 2 Sockets, 12 Cores per Socket, 2 Threads per Core
GPU	2 x NVIDIA A10, 24 GiB VRAM each
RAM	256 GiB System Memory
Storage	High-speed SSD
Network	NetXtreme BCM5720 Gigabit Ethernet PCIe, 1Gbit/s

Table 6.1. *Hardware characteristics of the computational environment*

For the execution of the experiments, a bash script named `run_screen_experiment.sh` was utilized to allocate specific CPUs for use. This script was configured to use 8 CPUs, ensuring that

[1] Deployment on a local computer with respectable resources could be achieved using the `run_simulation` utility of Flower, which optimizes load management via Ray, a software designed for resource management and process orchestration. This is accomplished by creating client batches that train without overwhelming the system's resources. Sequentially, all client batches are executed before aggregating results in a synchronous learning scenario. However, this approach is unsuitable for our implementation, as it would result in prolonged waiting periods for clients before aggregating their results. Our implementation of FDA utilizes synchronization of all clients on a per-batch basis, making this approach incompatible.

the computational requirements were met without overburdening the system. In contrast, the experiments demanded significant GPU resources, necessitating the activation of memory growth in TensorFlow to maximize the available GPU capacity. The GPU was heavily utilized for training processes on the client side and for centralized evaluation on the server side, ensuring optimal performance.

The aforementioned script executes the `run_experiments.py` Python module, which iterates over different experiment configurations. Each iteration of the loop represents a distinct experiment. During this process, four screen sessions are instantiated: one for monitoring `run_experiments.py`, one for overseeing `server_main.py`, one for supervising `main.cc` for the NS3 module, and one for managing `client_main.py`, which includes all client sessions in separate windows. This setup ensured that the experiments were conducted in a controlled and efficient manner, maintaining uninterrupted execution even if the connection to the server was lost.

6.3 Experiments Configuration

Before delving into the specific experimental analyses, this section delineates the configurations and baseline parameters employed in our study. To fully comprehend the experimental setup, we present tables for each component of the implementation. Bold values denote the baseline template used for all experiments, while the other values indicate alternative configurations tested and included in the research findings.

Firstly, the FL Orchestrator setup is detailed in the following table, showcasing the parameters utilized in the experiments. These encompass the range of client populations, datasets and ANNs, data distributions, and synchronization algorithms employed. These parameters serve as a reference point for all subsequent experiments:

Parameter	Value
Client Population	{5, 10, **15**, 20}
Dataset	{MNIST, **FashionMNIST**, CIFAR10}
ANN Architecture	{LeNet-5, VGG16, ResNet50, **Dataset's Default** }
Optimizer Learning Rate	0.001
Data Distribution	{**IID**, Non-IID}
Non-IID Templates	{**None**, Low Heterogeneity, Medium Heterogeneity, High Heterogeneity}
Synchronization Method	{**Naive FDA**, Online SyncSGD}
Mini Batch Size	64
Simulated WiFi Connection	{Weak Wifi, Medium Wifi, **Fast Wifi**}

Table 6.2. *Federated Learning orchestrator parameter configurations*

Similarly, the following table details the parameter selections for the Naive FDA synchronization technique. It includes hyperparameters used to tune the frequency of communication, with various epochs, thresholds, and discount factors associated with different experiments.

(Dataset, ANN)	Epochs	Threshold	Threshold Discount Factor
(MNIST, LeNet-5)	40	3	0.98
(MNIST, Default)	40	8	0.98
(FashionMNIST, LeNet-5)	40	10	0.99
(FashionMNIST, Default)	40	{**80**, 100, 120, 140}	0.99
(CIFAR10, Default)	50	300	0.987

Table 6.3. *Naive FDA experiment configurations*

Lastly, we outline the three diverse templates for communication utilized by NS3. These templates simulate a wide range of network conditions, including a deprecated WiFi interface for

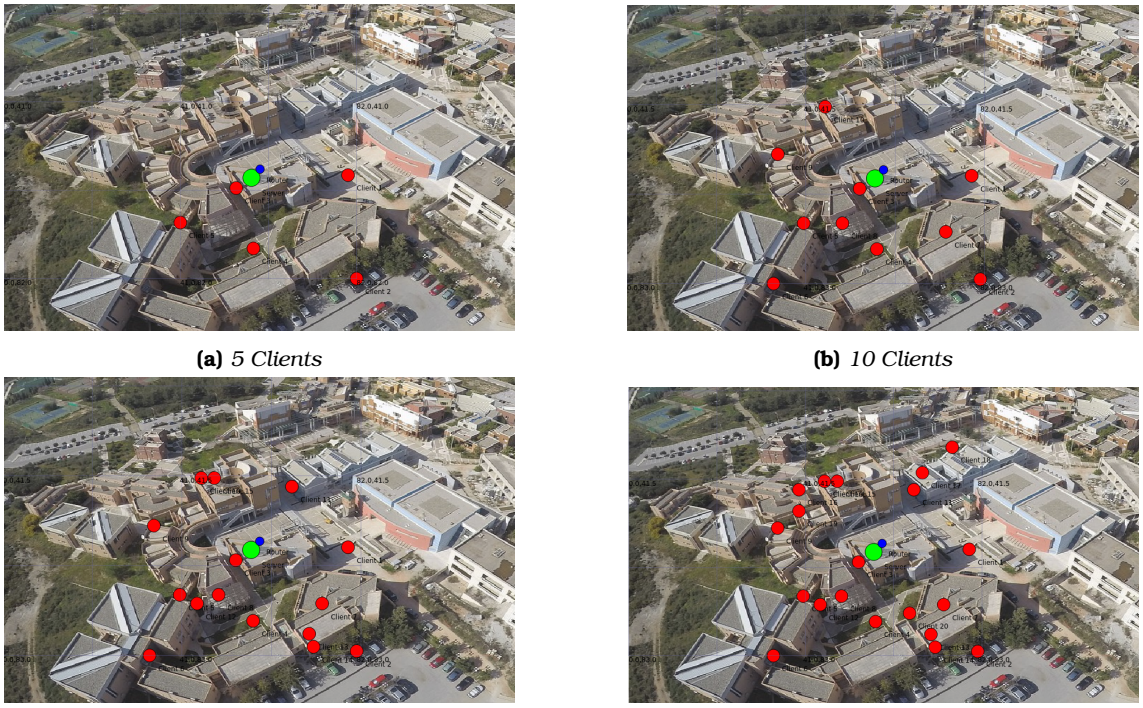
slow connections, a 2.4GHz WiFi connection for medium-speed scenarios, and a modern 5GHz WiFi of the 6th generation for fast transmissions.

The setups mostly share the same parameters, differing in only a few key aspects. Specifically, the primary differences are the WiFi standards used, the signal strength of the transmission, and the Remote Station Manager. Elucidating on that, the first is the primary factor determining the maximum achievable throughput of the network, the latter determining the maximum transmission range, and the third for dynamically adjusting transmission speeds MinstrelHighTransfer (MHT) is more effective in rapidly adjusting transmission rates in high-speed conditions compared to Adaptive Auto Rate Fallback (AARF), which is more suitable for stable but slower connections.

Parameter	**Weak WiFi**	**Medium WiFi**	**Fast WiFi**
Routing Protocol	Nix Routing		
Traffic Type	TCP		
Propagation Delay Model	ConstantSpeed		
Propagation Loss Model	LogDistance		
Server Data Rate	800 Mbps		
Packet Size	1024 bytes		
Error Rate Model	YansErrorRateModel		
Client Data Rate	Dynamic		
WiFi Standard	802.11g	802.11n_2.4GHz	802.11ax_5GHz
Signal Power	11.0 dBm	16.0 dBm	19.0 dBm
Remote Station Manager	AARF	MinstrelHighTransfer	MinstrelHighTransfer

Table 6.4. *Network simulation parameters for various WiFi configurations*

Furthermore, the client star topologies for 5, 10, 15 and 20 clients will be specified as the visualization of topologies will make it easier to conceptualize the implementation:



(a) *5 Clients*

(b) *10 Clients*

(c) *15 Clients*

(d) *20 Clients*

Figure 6.1. *Client topologies for different client populations generated with NetAnim*

The experimental topologies were designed to mirror a real-life scenario, using a part of the University's campus as the backdrop. In these diagrams, the clients are represented in red, the

server in green, and the router in blue. The topology creation process is noteworthy. The server is connected to the router via Ethernet, both situated in the same room, ensuring a stable and high-speed connection. The clients, however, are all connected to the router through WiFi, which means that the same network manages all the send and receive operations.

6.4 Findings and Analysis

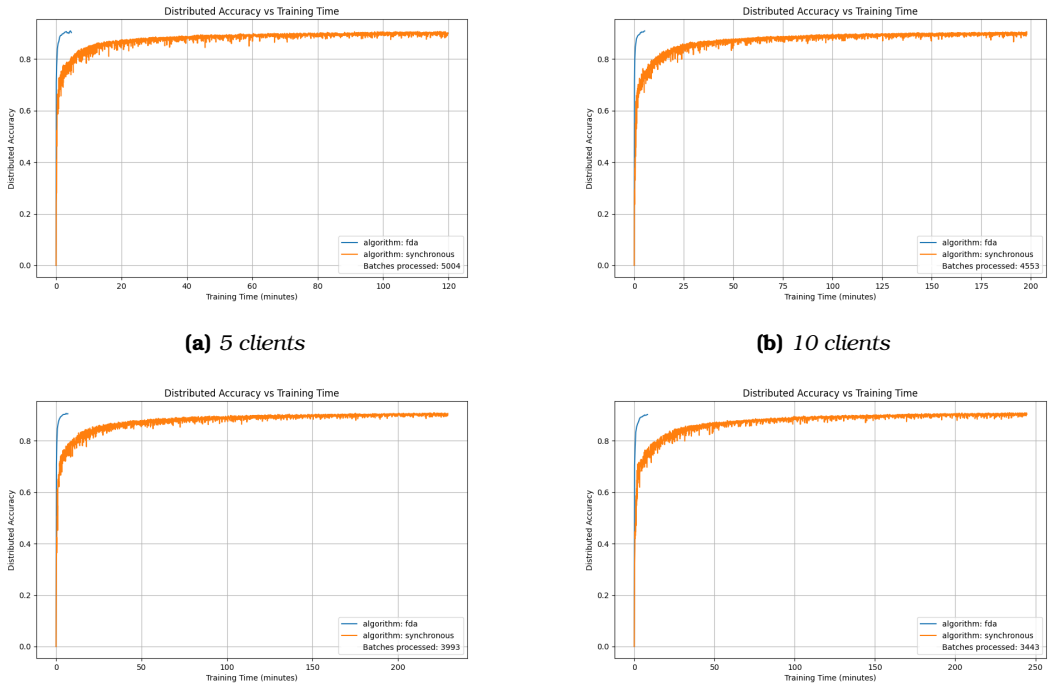
6.4.1 Comparison of FDA vs. Online SyncSGD Approach

Algorithm Convergence

To assess the learning efficiency, we compare the learning time and accuracy achieved by the FDA synchronization technique and the Online SyncSGD approach. To ensure a fair comparison, we measured the performance of both algorithms based on the same number of processed batches. We have meticulously recorded the process over 40 communication rounds for FDA and an equivalent number of rounds for Online SyncSGD, processing the same amount of data. However, for clearer visual comparisons, the results are displayed over a depth of 20 rounds.

Each subfigure provides a visualization of the distributed accuracy over time for both FDA and Online SyncSGD, evaluated in configurations with 5, 10, 15, and 20 clients respectively. The experimentations were conducted using the baseline configuration described in tables 6.2 and 6.3, with the minor adjustment of utilizing Lenet-50 as the ANN as it was the smallest providing swift execution in the long-running simulations of Online SyncSGD.

This detailed representation enables a comprehensive side-by-side comparison, highlighting the performance differences between the two synchronization techniques across various client populations.



(a) *5 clients* **(b)** *10 clients*

(c) *15 clients* **(d)** *20 clients*

Figure 6.2. *Accuracy over Time plots comparing FDA with OSyncSGD on diverse client populations*

The plots in Figure 6.2 reveal several key insights into the performance of the FDA and Online

SyncSGD synchronization techniques:

- **Convergence Speed:** The convergence speed of FDA is significantly greater than that of Online SyncSGD. This is evident from the steeper ascent in the accuracy curves of FDA across all client configurations. The faster convergence of FDA implies that it can achieve high accuracy levels in a shorter amount of time compared to Online SyncSGD.

- **Final Accuracy:** Both synchronization techniques achieve similar final accuracy levels, with Online SyncSGD slightly outperforming FDA by a negligible margin of approximately 0.67% in the worst case. This minor difference suggests that although Online SyncSGD may reach a marginally higher peak accuracy, the overall practical impact is minimal given its much slower convergence rate. It is important to note that the FDA executions were not fine-tuned for optimal performance, as the same setup was used across all experiments for consistency.

- **Scalability:** The scalability of both techniques is demonstrated by their consistent performance across different client populations. FDA maintains its rapid convergence advantage regardless of the number of clients, showcasing its robustness and scalability in handling varying client configurations. Improved performance is anticipated with an increase in the number of clients. However, this is not directly evident from fig. 6.2 and table 6.5 due to the significantly lower number of batches processed as the number of clients increases under the same conditions. For example, the setup with 5 clients processes 3075 more batches [2] than the setup with 20 clients, resulting in a minor accuracy increase of 0.64%. This, in itself, indicates the superior performance potential with an increased client population as the same accuracy is almost achieved in roughly a halved amount of batches.

- **Stability of Learning:** The stability of the learning process can be inferred from the smoothness of the accuracy curves. FDA exhibits stable learning behavior with less fluctuation in accuracy over time, indicating a more consistent and reliable learning process compared to Online SyncSGD.

In summary, the FDA synchronization technique exhibits a superior convergence speed and comparable final accuracy to Online SyncSGD. Its scalability and stability further underscore its effectiveness as a synchronization method in federated learning environments. The visualizations provided offer a clear depiction of the advantages of FDA, making a compelling case for its adoption in scenarios demanding efficient and robust learning across distributed clients.

Communication vs. Computation Times

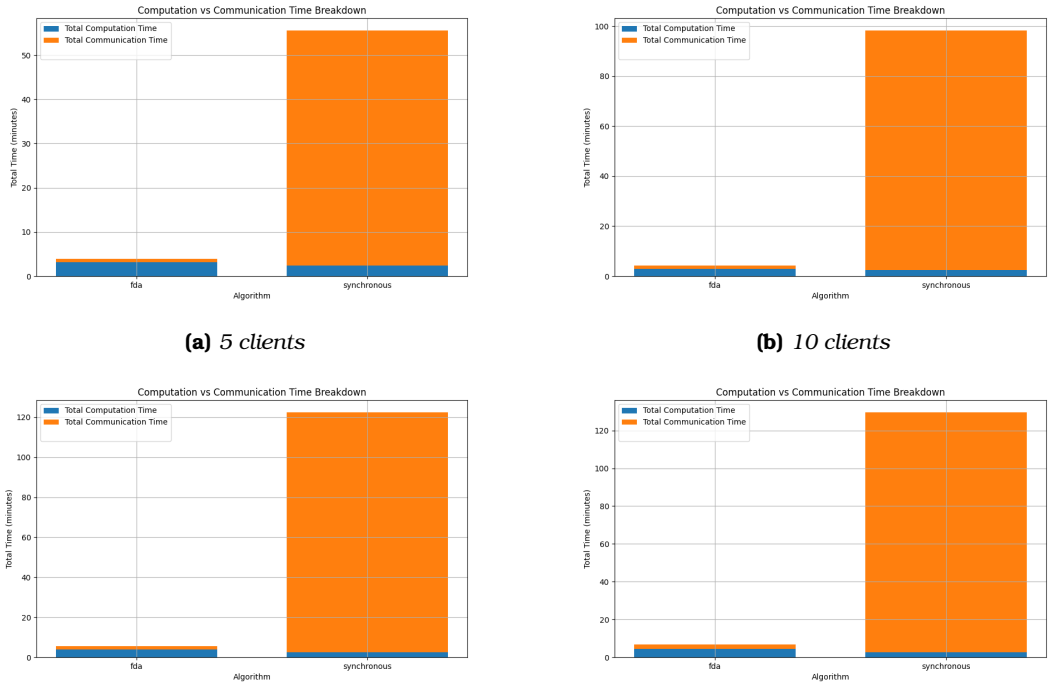
To further understand the efficiency of the FDA synchronization technique compared to the Online SyncSGD approach, we analyze the breakdown of computation and communication times for both algorithms. This analysis provides insights into the time distribution between computational tasks and the overhead caused by communication.

Table 6.5, with stats regarding the experiment is provided for better realization of the visualized values, before proceeding with diagrams. The stacked bar diagram in subfigures of 6.3 illustrates the total computation time (Blue) versus the total communication time (Orange) for both FDA and Online SyncSGD. Additionally, the subfigures of 6.4 display proportions of downlink (Blue), uplink (Green for FDA and Orange for OSyncSGD) and exclusively in the case of FDA, the RTC monitoring times (Orange) to better demonstrate the contribution of each part and its importance for the experiment's communication overhead.

[2] Batches processed are included in the legend of fig. 6.2 subfigures

Stat	5 Clients		10 Clients		15 Clients		20 Clients	
	FDA	Sync	FDA	Sync	FDA	Sync	FDA	Sync
Max Accuracy	0.9087	0.9101	0.9070	0.9088	0.9060	0.9083	0.9045	0.9078
Avg Communication Time(s)	47	3192	79	5754	105	7193	134	7618
Avg Computation Time(s)	188	142	175	141	228	156	273	159
Avg Downlink Time(s)	0.55	0.60	1.29	1.18	1.56	1.50	1.71	1.84
Avg Uplink Time(s)	0.09	0.08	0.15	0.17	0.35	0.31	0.59	0.51
Avg Rtc Monitoring Time(s)	1.72	Nan	2.50	Nan	3.35	Nan	4.42	Nan
Avg Round Time(s)	13.97	1.45	15.72	2.65	20.63	3.51	24.81	4.32

Table 6.5. *Computation and Communication related stats*



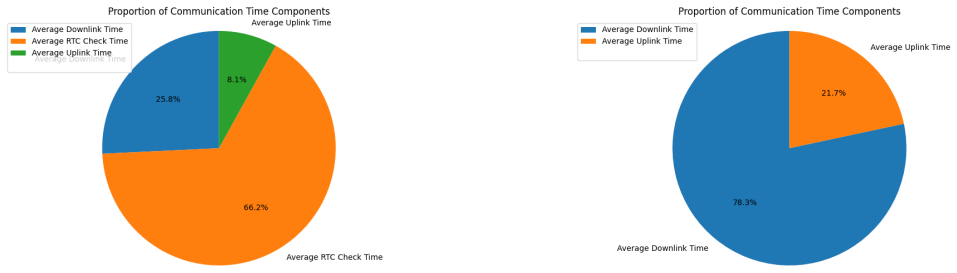
(a) *5 clients*

(b) *10 clients*

(c) *15 clients*

(d) *20 clients*

Figure 6.3. *Stacked bar diagrams comparing the Computation and Communication time contributions for FDA and OSyncSGD over varying client populations*



(a) *FDA*

(b) *Online SyncSGD*

Figure 6.4. *Pie diagrams comparing the communication time decomposition for FDA and OSyncSGD with 20 clients*

Understanding the breakdown of the communication-to-computation ratio is crucial in evaluating the overhead involved in FL processes. The results for comparing the communication-to-

computation ratio across the four client setups is exhibited as follows:

- *Computation to Communication Ratio:* The breakdown of training time into computation and communication times highlights the advantages of FDA over SyncSGD. Let the computation to communication ratio be denoted by A. For FDA, A ranges from roughly 2 to 4, while for SyncSGD, it ranges from 0.1 to 0.3. This significant difference indicates that FDA spends more time on computation relative to communication, whereas SyncSGD's overhead is dominated by communication.

- *Communication Overhead Decomposition:* Both FDA and SyncSGD exhibit similar uplink and downlink times per synchronization, as seen in table 6.5. However, the sheer number of synchronizations in SyncSGD leads to substantial communication overhead. RTC monitoring additionally burdens FDA, representing roughly 66% of its communication time. Despite this, taking the example of 5 clients (fig. 6.2a), FDA conducts only 20 model updates compared to SyncSGD's 4670. Thus, while the average round time is 12.52 seconds longer for FDA (table 6.5), SyncSGD performs 4670 more model communications.

  This discrepancy would only worsen with larger ANN models, where downlink and uplink times would overshadow the RTC monitoring time. The small size of LeNet (under 280 KB) makes RTC monitoring seem more significant, but with larger models, the frequent transmission of small integers would be negligible compared to model communications. Therefore, FDA's approach of fewer, more substantial communications results in a more favorable computation-to-communication time ratio, emphasizing its efficiency in distributed training.

- *Scalability influence:* While increasing the number of clients, a slight increase in the timing data can be observed in table 6.5. Specifically, both uplink and downlink times show a steady rise due to the increased network traffic. As more data is transferred simultaneously, slight delays become inevitable. This trend extends to other timing metrics as well.

  Another notable observation is the decrease in the computation-to-communication ratio with the addition of more clients. An increase in the number of participating clients leads to more time spent in communication, thus impacting overall efficiency. As the communication load grows, the proportion of time dedicated to computational tasks decreases, highlighting the challenges of scaling in distributed systems.

6.4.2 Impact of Varying Thresholds

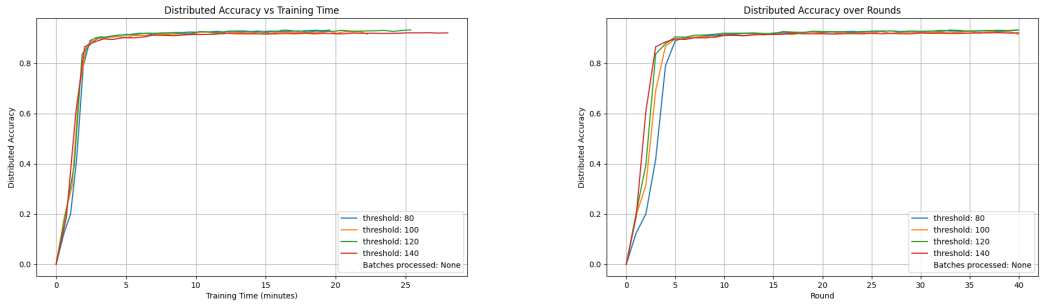
Algorithm Convergence

Following the logic introduced in the previous section, we begin by evaluating the convergence time of the FDA algorithm under varying threshold values, denoted as Θ. This evaluation aims to identify the strengths and weaknesses of different threshold settings and to determine the optimal value that best meets the objectives of the approach.

The experiments presented in this section adhere to the baseline characteristics outlined in tables 6.2 and 6.3. Each experiment extends over 40 synchronization rounds, employing diverse threshold values. For clarity, it is important to note that these experiments use the Default ANN implementation designed specifically for the FashionMNIST dataset. The tested threshold values are 80, 100, 120, 140, selected to suit the ANN architecture.

The FDA algorithm demonstrates robustness by achieving similar accuracy across a range of threshold values. However, a closer examination reveals some distinct trends:

- *Initial Convergence:* Higher threshold values accelerate convergence in the early stages of training. This is because higher thresholds allow clients to process a greater number of data batches



(a) *Accuracy over Time* **(b)** *Accuracy over Rounds*

Figure 6.5. *Combined plots of Accuracy over Time and Rounds for various RTC threshold assignments*

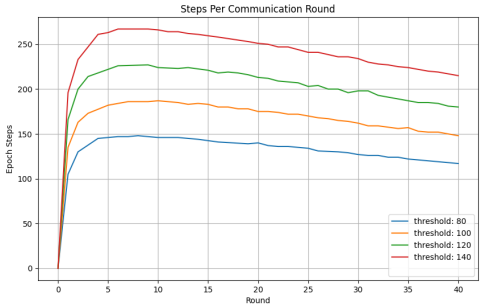


Figure 6.6. *Steps per communication round*

before synchronization, leading to quicker initial insights due to reduced communication frequency. Essentially, increasing the threshold correlates with fewer communication rounds, as the model variance threshold requires more time to be surpassed.

- *Sustained Performance:* Although higher thresholds benefit from rapid initial convergence, this advantage diminishes over time. Algorithms with more frequent communication rounds (lower thresholds) gain from more accurate variance monitoring, maintaining a precise estimate of the global model. This is evident after approximately 5 rounds, as shown in the corresponding figure, where algorithms with smaller thresholds converge more rapidly.

- *Training Completion:* Algorithms with lower thresholds complete the predefined 40 rounds more swiftly. The increased frequency of communication facilitates more consistent updates and adjustments, enabling faster overall convergence. The reason for that is that they execute fewer steps per communication round meaning restricted local updates without deviating far from the initial estimation of the global model. Thus, rounds complete prematurely in comparison with higher thresholds as indicated by 6.6.

- *Final Convergence:* Ultimately, convergence to high accuracy is achieved more quickly with smaller thresholds. Frequent updates help maintain a closer alignment with the global model, thereby enhancing learning efficiency and reducing the time required to achieve high accuracy levels.

- *Round Steps:* The number of steps completed per round initially increases and peaks somewhere after 5 rounds. The reasons for this deceleration in the number of steps are twofold. Firstly, as the global model accumulates more knowledge, the variations in local model updates from

the model sent at the start of the round are significantly reduced. This reduction in variation leads to fewer steps being required to reach the threshold for synchronization. Secondly, a discount factor is applied to the threshold value at each round. For our experiments, we chose a discount factor that is not optimized but is intended to gradually reduce the threshold value so that by the end of training, the threshold will be approximately 60% of its original value. This strategy aims for bigger local training steps to accumulate knowledge faster by utilizing maximal computation resources at the start, while reducing steadily the threshold for closer tracking of the variance and optimized results while the model gets smarter. This approach balances the need for rapid initial learning with the necessity of maintaining accurate global model updates, ensuring that the training process remains efficient and effective throughout its duration.

In summary, while higher thresholds provide a rapid initial boost by allowing extensive local updates, the frequent communication enabled by lower thresholds proves more effective in sustaining convergence and achieving high accuracy in the long run.

Communication Vs. Computation Times:

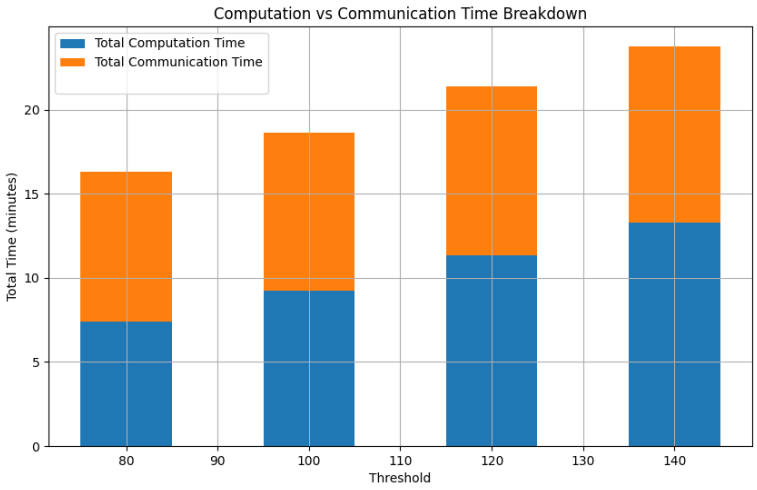


Figure 6.7. *Stacked bar diagram representation of Computation and Communication times*

- *Overall Time:* Raising the threshold value (Θ) results in a proportionate increase in the overall learning time, as observed in fig. 6.7. Specifically, incremental updates to the threshold value, applied in intervals of 20, led to a similar linear increase in the total round time. The stacks in the graph display almost equal step sizes, indicating a direct correlation between the threshold value and the overall learning duration. This suggests that higher threshold values, which reduce the frequency of communication, ultimately extend the time required for the learning process to complete.

- *Decomposition commentary:* Increasing the threshold value (Θ) directly impacts the balance between computation and communication times. Specifically, as the threshold value increases, the proportion of time spent on computation relative to communication also increases. This shift occurs because higher threshold values reduce the frequency of communication, allowing more data to be processed locally before synchronizing with other nodes. Consequently, the system spends more time on computation tasks, leveraging local processing capabilities, and less time

on the overhead associated with frequent communication. This adjustment can enhance overall efficiency by minimizing the interruptions caused by communication, particularly in scenarios where communication latency is significant.

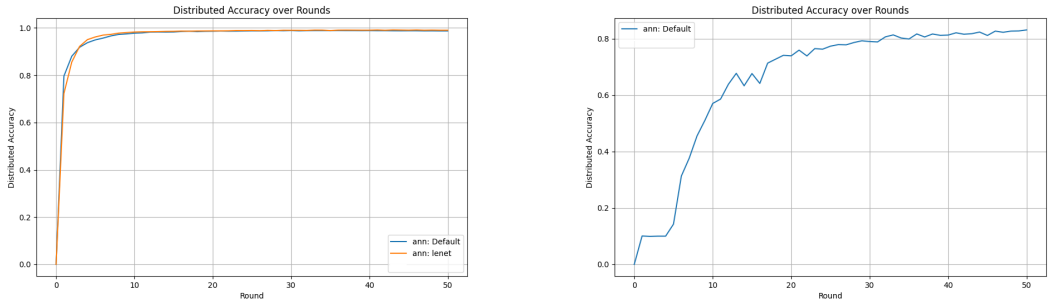
Taking everything into consideration, smaller thresholds are preferable for maintaining a more accurate estimation of the globally accumulated knowledge. This is due to the increased frequency of communication, which ensures that model updates are promptly shared among nodes, leading to higher accuracy. Such a selection would be optimal in scenarios where nodes are interconnected with fast and stable connections, and latency is not a primary concern. The primary focus in these scenarios is on achieving the maximal possible accuracy.

It's worth noting that in the extreme case of setting the threshold to zero, the FDA approach effectively reduces to the simple OSyncSGD implementation used in our comparisons. This means that the benefits of the FDA approach are entirely negated, as the model updates would occur as frequently as in the OSyncSGD method, eliminating any advantage gained from the dynamic synchronization.

On the other hand, higher thresholds reduce the frequency of communication, thereby shifting the balance towards more local computation in the expense of communication . This can be beneficial in environments where communication overhead is a significant concern, such as in networks with high latency or limited bandwidth. By allowing nodes to process more data batches locally before synchronizing, higher thresholds can reduce the overall communication load and improve efficiency in such scenarios. However, this comes at the cost of potentially slower convergence to the desired accuracy levels due to the less frequent updates.

6.4.3 Diversity of Datasets and ANN Architectures

This section evaluates the robustness and adaptability of the FDA algorithm across some further datasets and artificial neural network (ANN) architectures. The objective is to demonstrate the generalizability of the results obtained in previous sections to other ANN architectures and datasets beyond the selected baseline combination. The setups presented here utilize the parameters outlined in table 6.2, and the results depict the accumulated accuracy over communication rounds.



(a) *MNIST tested on Default ANN and LeNet-50* **(b)** *CIFAR-10 tested on Default ANN and LeNet-50*

Figure 6.8. *FDA tested on diverse ANN-dataset combinations for MNIST and CIFAR-10*

The findings presented in this section confirm that the FDA algorithm's effectiveness extends across different ANN architectures and datasets. Specifically:

- **MNIST Dataset:** As illustrated in fig. 6.8a, FDA achieves optimal performance with minimal communication rounds on the MNIST dataset. Both the *LeNet-50* and *Default* architectures reach an accuracy of 98% in fewer than ten communication rounds and continue to improve

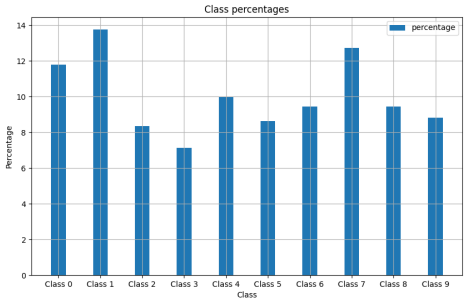
beyond 99% at a slow pace. This rapid convergence demonstrates the algorithm's pure learning capabilities that do not fall far from what a centralized approach would accomplish, reaching rapidly the ann's learning limits in this simple example.

- **CIFAR-10 Dataset:** In fig. 6.8b, the FDA's application to the CIFAR-10 dataset using the *Default* ANN architecture shows substantial learning progress, converging to over 80% accuracy. Although the process exhibits some stability issues, indicated by oscillations in the accuracy graph, these can be attributed to the basic nature of the ANN architecture used. It is expected that results will improve with a more fine-tuned implementation. Nonetheless, this was outside the primary research objectives of this dissertation, and this architecture was selected for its compact size [3].

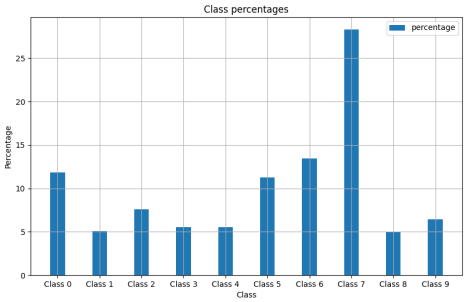
6.4.4 Handling Non-IID Data Distributions

The robustness and adaptability of the proposed system under non-IID data distributions are crucial for its practical application, as real-world learning is primarily conducted under such conditions. In this section, we evaluate how the Naive Functional Dynamic Averaging (FDA) synchronization technique performs when faced with non-IID data. By testing different levels of data heterogeneity, we aim to demonstrate the system's ability to maintain high performance despite adverse conditions.

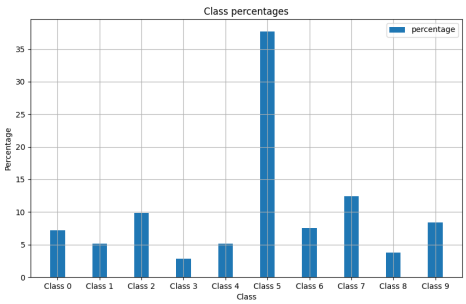
To achieve this, we conduct a series of experiments using three bias templates representing varying degrees of non-IID data: small, medium, and large heterogeneity. These experiments are executed in scenarios with a restricted number of clients (5) and a larger client pool (15) to analyze the impact of client population on performance. Before presenting the results, the following plots showcase random examples generated by the utilized data distributions.



(a) *Low Heterogeneity*



(b) *Medium Heterogeneity*



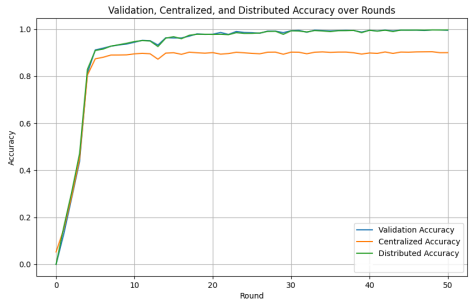
(c) *High Heterogeneity*

Figure 6.9. *Class percentage examples of the non-i.i.d templates distributions*

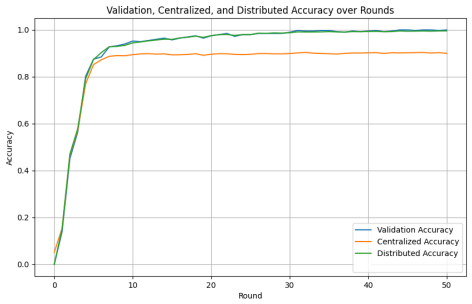
[3] The reason for that will be outlined in the thesis challenges and future work sections.

Achieved Accuracies:

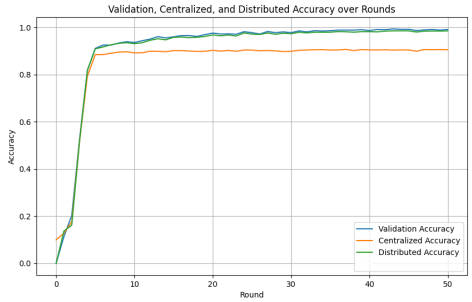
The results of the experiments will now be presented. The baseline conditions for the learning process, as detailed in tables 6.2 and 6.3, are utilized. The first set of results depicts the local validation accuracy, the distributed accuracy and the centralized accuracy for various heterogeneity levels and client populations:



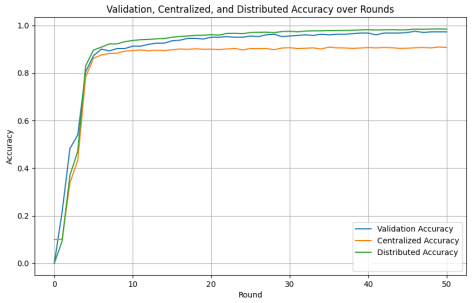
(a) *Low Heterogeneity with 5 clients*



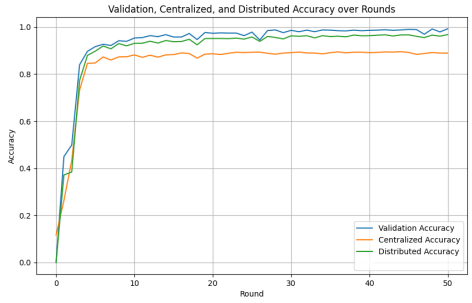
(b) *Low Heterogeneity with 15 clients*



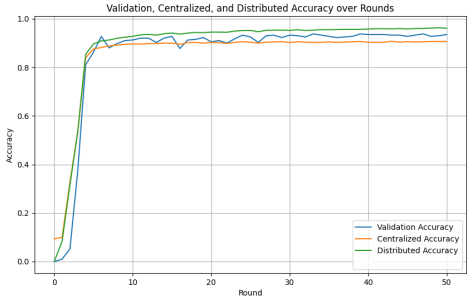
(c) *Medium Heterogeneity with 5 clients*



(d) *Medium Heterogeneity with 15 clients*



(e) *High Heterogeneity with 5 clients*



(f) *High Heterogeneity with 15 clients*

Figure 6.10. *Plots containing combined: local validation accuracy, distributed accuracy and centralized accuracy for varying heterogeneity levels and client populations*

To avoid confusion the three depicted categorizations of accuracy will be clarified:

- *Local Validation Accuracy:* This represents the accuracy of the global model when tested locally on the validation set of a client participating in training. Notably, the local validation set of the clients is comprised of the same distribution as the training dataset.

- *Distributed Accuracy:* This represents the perceived accuracy of the FL system. In most FL scenarios, a test set containing all classes is non-existent on the server. To validate its knowledge, the server aggregates the evaluation results of the participating clients, which are the aforementioned validation accuracies.

- *Centralized Accuracy:* For referencing, in our case, we have assigned the FL server with an IID test set for centralized evaluation.

The experiments on diffrerent heterogeneity levels will now be discussed:

- *Individual Impact:* A significant observation related to the figures in 6.11 pertains to the nature of Federated Learning and, specifically, the aggregation algorithm of FedAvg. Notably, the distributed accuracy of the system in scenarios with 5 clients shows a close correlation with the fluctuations in validation accuracy. This phenomenon is more pronounced in cases with greater data heterogeneity. As previously discussed, the individual contribution of a single client is magnified when fewer clients participate in training. For instance, in a scenario with 5 participating clients, the contribution of a single client accounts for 20% [4] when there are no dropouts due to network issues or more otherwise. In contrast, the effect of a single client's contribution is significantly diminished when more clients are involved.

- *Distributed Accuracy Level:* The distributed accuracy, while following the trends of the local validation accuracies, becomes significantly greater than that observed in the IID-tested counterpart discussed in section 6.4.2. However, this increase does not necessarily translate to better real-world performance. In previous experiments, the maximal accuracy achieved was roughly 92%, whereas in this scenario, the result converges to almost perfect accuracy. This phenomenon occurs because, with higher bias, the network tends to overfit more on the locally dominant classes. When different clients have varying dominant classes, the aggregated global model starts to recognize these classes more effectively, improving performance on them. Consequently, when the global model is validated using the clients' biased validation sets, which contain these exact dominant classes, the perceived distributed accuracy of the FL system increases substantially. However, this perceived knowledge does not necessarily indicate a generalized capability of the system to perform well on any random sample. The observed increase in distributed accuracy is thus a reflection of overfitting to the biased local data rather than an improvement in the model's overall generalization ability.

- *Centralized Accuracy:* The centralized accuracy metric reveals the actual capability of the Federated System to identify all classes accurately. Observations indicate that the system's performance under non-IID conditions is slightly less effective than in the IID scenario. However, the drop in accuracy is not substantial. For instance, in the scenario with 5 clients, the decrease in centralized accuracy is almost 1%, as shown in 6.5. This suggests that while the system performs well, there is a minor compromise in its ability to generalize across all classes under non-IID conditions.

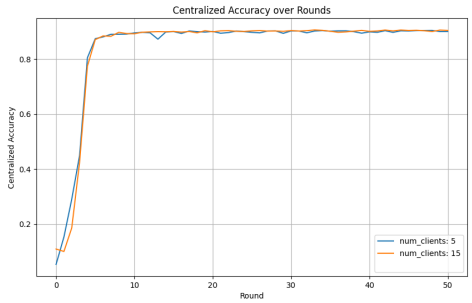
In summary, the robustness and adaptability of the proposed system under non-IID data distributions are critical for practical application. The experiments demonstrate that the Naive Functional Dynamic Averaging (FDA) synchronization technique can maintain high performance despite these adverse conditions. Overall, these experiments highlight the system's strengths and areas for improvement when dealing with non-IID data distributions, underscoring its potential for real-world federated learning applications.

Scalability Effect:

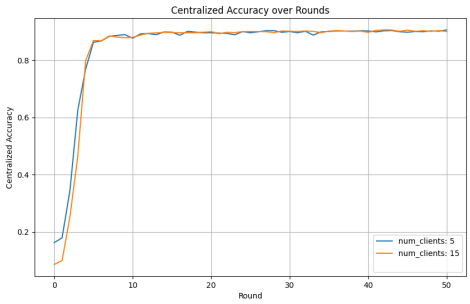
The second set of experiments on non-iid data distributions aims to investigate the effect of an increased client pool participating in training in comparison with a more restricted one. The following experiments demonstrate in a common graph the combined centralized accuracy of the

[4] Same number of samples is processed by all the clients as we maintain per batch synchronization through FDA

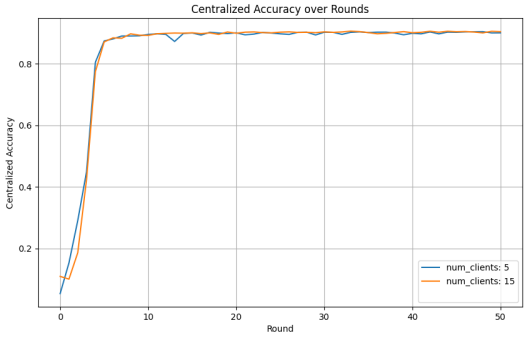
model when training upon the aforementioned setup with 5 clients and 15, over the training rounds.



(a) *Low Heterogeneity*



(b) *Medium Heterogeneity*



(c) *High Heterogeneity*

Figure 6.11. *Plots containing combined the centralized accuracies for 5 and 15 clients for varying heterogeneity levels*

- *Stability:* The inclusion of more clients in the training process stabilizes the curve oscillations, as the contribution of more learning nodes results in a more evenly distributed load for each client. Each update committed to the global model reflects the processing of more total data spread across multiple nodes, producing a steadier increase in accuracy without fluctuations.
- *Effect of Heterogeneity:* As heterogeneity increases, the centralized accuracy tends to stabilize at a slightly lower level, indicating that higher data variability across clients can slightly hinder the model's ability to generalize perfectly. Nonetheless, it needs to be stated that the Federated Learning process demonstrated significant robustness retaining an accurate estimate at around 90% regardless of the heterogeneity's level.

In closing analysis, increasing the number of clients participating in training leads to a more stable and consistent learning process, even with high data heterogeneity. However, this increased heterogeneity slightly affects centralized accuracy, indicating a trade-off between data diversity and model performance.

The system performance tests use a methodology where each client's partition ID is used as a seed in the Python random function. This ensures each client receives a unique, client-specific dataset through a random distribution, inherently introducing client data heterogeneity.

This method does not explicitly control or measure overall system heterogeneity. Although creating an optimal non-IID scenario is not the primary objective of this thesis, observing performance under these conditions provides valuable insights. As more clients are involved, the combined data may approximate an IID distribution, smoothing out individual biases.

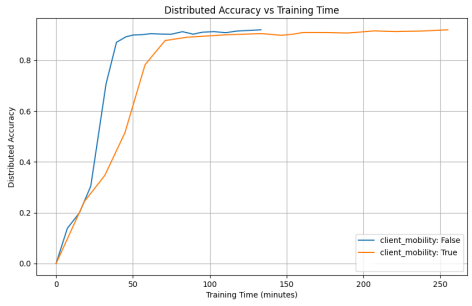
Overall, the system demonstrates robust performance and adaptability in handling client-based data heterogeneity, despite the inherent challenges of federated learning with non-IID data.

6.4.5 Diverse Network Condition Scenarios

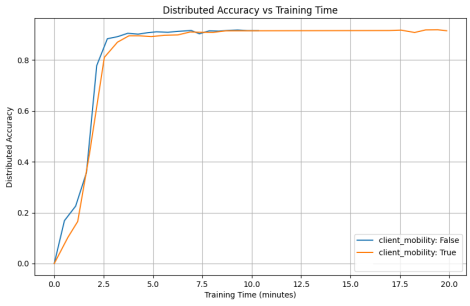
Real-world applications of FL are often subject to diverse and challenging network environments, which can significantly impact the performance and reliability of the learning process. Therefore, it is crucial to assess the robustness and adaptability of our FL system across different networking scenarios.

To simulate realistic conditions, we conducted experiments alternating between mobile and stationary clients while utilizing weak, medium, and fast WiFi connections. This approach allows us to understand how varying network quality and client mobility affect the FL process. The following set of experiments presents the accuracy convergence over time over 20 rounds of communication. The parameters for the federated system and the fda synchronization follow the baseline configurations of tables 6.2 and 6.3.

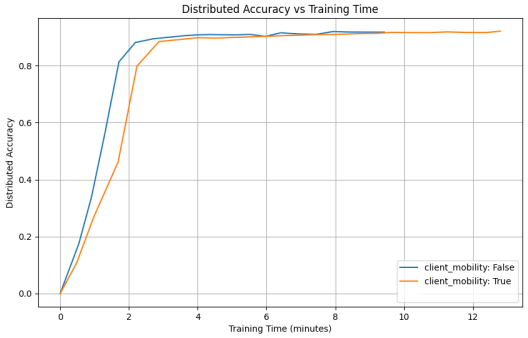
Convergence:



(a) *Weak WiFi Setup*



(b) *2.4 GHz WiFi Setup*



(c) *5GHz WiFi 6 Setup*

Figure 6.12. *Accuracy over Training Time plots, comparing stationary and mobile clients under varying network conditions*

- *Execution Time:* The execution time is consistently longer for mobile clients across all scenarios. This phenomenon is due to the fact that mobile clients while moving, may experience diminished WiFi connections compared to stationary nodes, which are positioned within the optimal communication range or at the maximum supported WiFi distance. Consequently, the time required to receive the model from the server or transmit it back can increase proportionally.

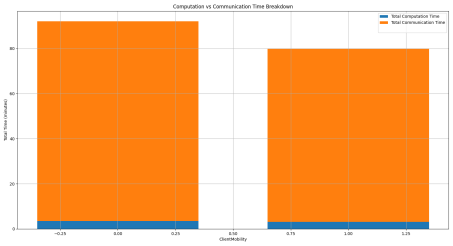
- *Convergence Accomplished:* A detailed analysis reveals two key insights regarding convergence. Firstly, when comparing convergence across different WiFi configurations, the achieved accuracy is similar, around 90%-91%, with marginal improvements in faster connections. Better WiFi connections benefit from fewer dropouts, allowing more clients to participate in the training process per round, thus contributing to slightly better results. Additionally, faster network conditions lead to quicker convergence as more clients participate in training, and less communication time is required. Secondly, within the same WiFi setup, stationary clients tend to achieve faster convergence. This is explained by the better network conditions provided by stationary clients, as previously mentioned.

- *Learning Stability:* An increased number of dropouts leads to a proportional decrease in the system's learning stability. This is because the individual impact of a client's learning contribution is magnified when the number of clients participating in training decreases.

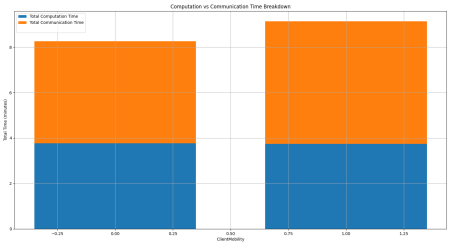
All in all, The findings indicate that mobile clients experience longer execution times due to diminished WiFi connections, while stationary clients achieve faster convergence and better learning stability due to fewer dropouts and more consistent participation in the training process. Improved WiFi configurations lead to marginally higher accuracies and faster convergence, highlighting the significant impact of network conditions on Federated Learning performance.

Computation to Communication Ratio:

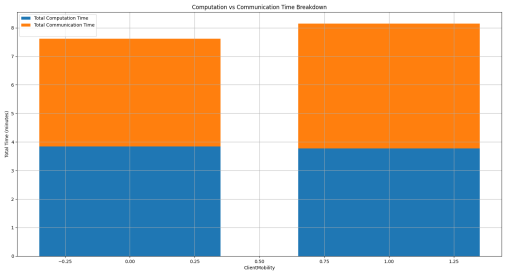
The results of the first set of experiments demonstrate the decomposition of total round time into computation and communication times, allowing us to monitor variations in the computation-to-communication ratio.



(a) *Weak WiFi setup*



(b) *2.4 GHz WiFi setup*



(c) *5 GHz WiFi 6 setup*

Figure 6.13. *Computation to Communication Time comparison in stacked bar diagrams comparing stationary and mobile clients under varying network conditions*

- *Computation Time:* The total average computation time changes by a negligible margin. This is because the average computation time per round is not significantly influenced by varying the

number of clients between 12 and 15 for example. On average, the time spent on computation remains consistent, regardless of the changes in the network conditions. That leads to a value close to 4 minutes over the entirety of client mobility and wifi template setups. with marginal differences due to the slight change in the number of participating clients.

- *Communication Time:* The total average communication time for the scenarios depicted in Figure 6.13a exhibits different behavior compared to those in Figures 6.13b and 6.13c. This discrepancy is primarily due to weak WiFi significantly slowing down as more clients participate, and in many cases, leading to an inability to communicate due to distance and decreased signal strength. As observed in Table 6.6, the average number of dropouts per round increases from 2.75 when clients are stationary to 4.2 when they are moving, a difference of approximately 1.5 dropouts per round. Consequently, fewer clients participate in the communication process, leading to an increase in the communication throughput of the remaining clients. This results in less average communication time when clients are moving and experiencing more dropouts.

  On the other hand, with the two better WiFi configurations, the situation changes. In these cases, client movement results in only a small increase in dropouts, roughly 0.2 per round. This indicates that the number of clients participating in the training process changes only marginally. However, although these clients can be served by the network, their communication time varies depending on whether they move closer to or further from the router. This variability introduces instability, resulting in more time being consumed in communication as clients adjust to fluctuating signal strengths and network conditions. This leads to longer average communication times under medium and fast WiFi conditions, even with relatively stable dropout rates.

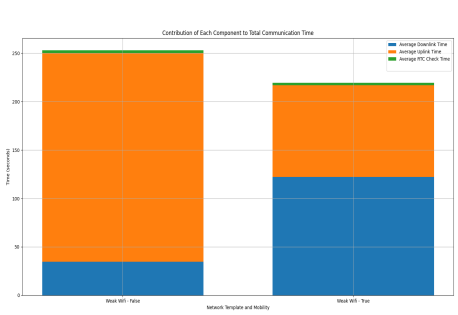
Overall, the computation-to-communication ratio adjusts according to the circumstances discussed. Increasing WiFi speed templates causes the ratio to proportionally increase, thereby cutting down communication costs. Conversely, client movement introduces more or less overhead in communication depending on whether the clients can maintain better network connections. In this analysis, it is unnecessary to mention computation time, as it remains almost constant across different scenarios. Thus communication is the determining factor of the fraction.

6.4.6 Communication Decomposition

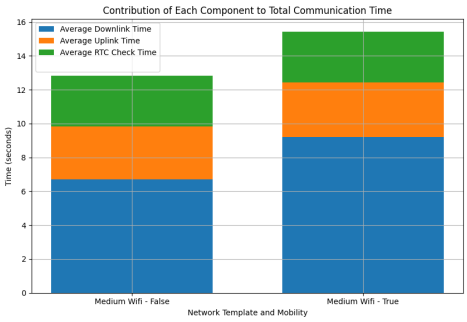
This section seeks to explain the communication time decomposition tested under the same conditions as before. Subsequently, a table with overall timing statistics is demonstrated along with a series of plots that facilitate the visualization of the data presented. The plots contain the segmentation of the overall training communication in downlink, uplink and RTC monitoring times.

Network Template and Mobility	AVG Throughput	AVG Dropouts	AVG Downlink	AVG RTC Monitoring	AVG Uplink	Total Communication
(Weak Wifi, Stationary)	1.06	2.76	34.57	3.10	215.27	252.94
(Weak Wifi, Mobile)	3.06	4.19	122.18	2.73	94.60	219.52
(Medium Wifi, Stationary)	14.97	1.90	6.71	3.01	3.12	12.83
(Medium Wifi, Mobile)	16.54	2.00	9.29	2.99	3.22	15.42
(Fast Wifi, Stationary)	32.92	1.86	5.51	3.00	2.26	10.77
(Fast Wifi, Mobile)	39.69	2.00	7.65	3.04	1.81	12.50

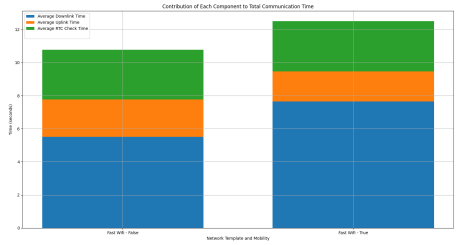
Table 6.6. *Communication Metrics computed for diverse network template and client mobility setups*



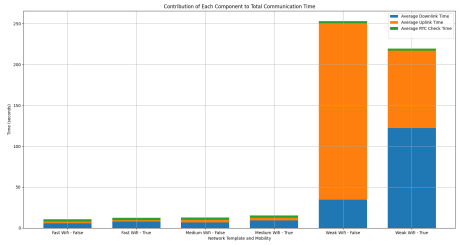
(a) *Weak WiFi setup*



(b) *2.4 GHz WiFi setup*



(c) *5 GHz WiFi 6 setup*



(d) *Common demonstration for all WiFi setups*

Figure 6.14. *Communication Time decomposition in stacked bar diagrams comparing stationary and mobile clients under varying network conditions*

- *Dropout Ratio and Average Throughput:* Across all experiments, the dropout ratio increases with client mobility. This increase is due to the uncertainty introduced by the random movement of clients, which may cause them to move outside the communication range and lose WiFi connection. This directly correlates with the overall increase in throughput when clients are moving. Fewer clients attempting to communicate due to the higher dropout ratio result in more available bandwidth for the remaining clients, who then complete the communication process more efficiently.

- *Downlink and Uplink Times:* The downlink and uplink times are influenced by the proximity of clients to the router due to random movement. When clients move closer to the router, downlink and uplink times decrease; conversely, these times increase when clients move further away.

- *RTC Monitoring Time:* The duration of RTC monitoring remains constant across different scenarios, as the tested conditions do not affect it. RTC monitoring is measured in real-time between the participating clients of the Federated System. This approach was chosen to minimize complexity, as it requires per-batch transmission and closely mirrors real-life conditions.

- *Weak WiFi Setup:* In the weak WiFi setup, communication times decrease when clients are moving. This is because client movement often leads to an inability to serve a substantial portion of them, resulting in better transmission speeds for the remaining clients who can now utilize a larger portion of the limited bandwidth available.

Part III

Epilogue

Chapter 7

Conclusions and Future Work

7.1 Research Conclusions

This thesis set out to explore and address the challenges inherent in Federated Learning (FL) environments, with a particular focus on integrating realistic network conditions and applying effective synchronization in distributed systems for Online Machine Learning (OML) scenarios. Traditional frameworks in this field often necessitate a deep understanding of distributed learning systems to orchestrate FL processes and typically fail to emulate realistic networking conditions during simulations.

This presents a significant oversight, as testing FL often requires immense resources for the virtual instantiation of numerous clients, with each client's machine learning process imposing a considerable computational load. Consequently, evaluations are often performed using powerful computing infrastructure under ideal network conditions. This creates a substantial discrepancy because, in practice, FL is typically conducted in adverse and unstable network environments. Despite this, an all-inclusive framework that enables the seamless design of FL systems capable of performing under varying simulated network conditions is notably absent from the field.

This research successfully addressed this problem by integrating the Flower framework[86] for FL orchestration with the NS3[87] network simulator, thereby creating a framework that can implement FL architectures with minimal effort due to the abstracted infrastructure of the tool, and a realistic networking environment for testing and development.

The framework also incorporated the Functional Dynamic Averaging (FDA)[16] synchronization technique, based on the theoretical underpinnings of the Geometric Method (GM)[17], to verify the infrastructure's effectiveness. The method has previously been tested and validated [16, 82, 83], but not within such a comprehensive framework that can highlight the impact of networking adversities. FDA's evaluation demonstrated significant improvements in communication efficiency by reducing the volume of communication and maintaining equivalent accuracy across various network conditions compared to the Online Synchronous SGD (OSyncSGD). This innovative approach effectively balanced the computation-to-communication ratio, proving to be both scalable and robust.

Both the implemented framework and the FDA synchronization technique were meticulously tested and evaluated, presenting key findings that underscore the efficacy, robustness, adaptability, and scalability of both the infrastructure and the synchronization method. Specifically, the experiments included a side-by-side comparison between FDA and OSyncSGD, highlighting the significant communication overhead mitigation performed by FDA across various client populations. Additionally, an effort was made to tune the hyperparameters related to the synchronization threshold for FDA. The adaptability of the framework was exhibited by extending the results to other artificial neural networks (ANNs) and datasets beyond the baseline approach. Last but not least, Robustness was demonstrated on non-IID data distributions, and the impact of networking

conditions was highlighted through diverse networking scenarios involving both stationary and mobile clients.

In conclusion, this research provides a novel and comprehensive solution for implementing and testing FL systems under network conditions that mirror real-life FL networks, with diverse throughput, latencies, and mobility models. The integration of realistic network simulations into FL frameworks represents a substantial step forward in the theoretical understanding of decentralized machine learning systems, enabling practitioners to monitor more accurately their system's performance through the simulation of dynamic and unpredictable environments before deployment. Whereas the synchronization technique of FDA proved once again that enhances significantly the communication efficiency while maintaining high accuracy, paving the way for future advancements in the field of Federated Learning.

7.2 Future Work

Building on the findings of this research, several avenues for future work are identified to further enhance the implemented framework and its capabilities. These recommendations include additional operations to demonstrate the current implementation's capabilities, further optimizations to increase the efficiency of the proposed infrastructure, and substantial additions to the framework's simulating capabilities.

7.2.1 Real World Verification of the Results

To validate the accuracy of the simulated metrics produced via the network and FL simulations, a real-life deployment of the proposed network setups is recommended. This could involve using mobile devices or Raspberry Pis as clients and a central server based in the university's laboratory to act as the Federated Learning server. Such a setup, while requiring careful structuring, would provide valuable insights into the proposed approach's potential by validating the simulated results. This real-world verification would also shed light on any discrepancies between the simulated environment and actual deployments, enhancing the framework's practical applicability.

7.2.2 Extending Heterogeneous Testing of FDA

The proposed approach currently generates heterogeneous data with varying bias for each client individually, ensuring each client is trained on a differently segmented dataset (client data heterogeneity). However, this does not guarantee a diverse distribution of data across the overall system (system data heterogeneity). Future work should explore scenarios where instances of some classes are overrepresented, creating an imbalance in the total processed data. This would provide insights into the system's performance in even more adverse and realistic scenarios, further testing the robustness of the FDA synchronization technique under extreme conditions.

7.2.3 Additional Testing on More Advanced ANNs

The datasets tested in the current implementation are of restricted size, with the largest tested ANN being the default implementation for CIFAR-10, which has a serialized size of roughly 50 MB. Extending the applicability of the simulator to larger ANNs used in the market for more extensive datasets would be a valuable next step. This would test the framework's capacity to handle larger and more complex neural networks, providing a more comprehensive evaluation of its performance and scalability.

7.2.4 Optimization Regarding NS3 Simulations

An important optimization for future work involves leveraging the deterministic nature of NS3 simulations. Without the inclusion of a seed during NS3 simulations, the results of a single simulation over the rounds will be the same, given the same settings. This characteristic can be used to hash the results of each setup and utilize them independently from the FL process. By doing this, the FL orchestrator and the network simulator would be delineated, allowing for faster results. Instead of simulating network parameters before conducting each round of FL, precomputed hashed results could be used, significantly reducing the time required for FL experiments.

7.2.5 Computation Capacity Simulation

To further enhance the realism and adaptability of the results, integrating the logic for simulating computational overhead, similar to how communication is currently calculated, is recommended. The current implementation could benefit from modeling the processing characteristics of GPUs, CPUs, and memory characteristics of computing units. Providing templates for the computation capacity of devices such as Raspberry Pis, mobile phones, or local computer stations would facilitate researchers in understanding the computational timing parameters related to training execution on devices with diverse computing capabilities.

The importance of this approach cannot be overstated. It offers a two-fold contribution. Firstly, it improves the realism of Federated Learning timing measurements, aligning them more closely with real-life scenarios, as typical FL setups often involve devices with lower computational capacity. Secondly, it eliminates the need for system isolation and real-time measurement, as all recorded metrics would be based on simulated values. This is particularly important in environments where multiple researchers may require computational resources, as it ensures the validity of timing data without the need for isolated experiments, thus maintaining the accuracy of results even in shared computational environments.

This approach would require utilizing linear regression techniques to parameterize the emulation process of devices based on specific configurable parameters[84]. To achieve this, comprehensive data on the computational performance of various devices, including CPUs, GPUs, and memory configurations, must be collected. Benchmarking tests can measure training and inference times for different neural network architectures, memory usage, and computational load distribution. Key parameters such as CPU clock speed, number of cores, GPU processing power, and memory size should be identified and correlated with performance metrics using linear regression models. These models can then be integrated into the FL simulation framework to estimate computation times for each client based on their hardware specifications. Regular validation and updates of these models with new performance data will ensure their accuracy and relevance, thereby enhancing the realism and reliability of the simulations in reflecting the performance of federated learning systems across diverse devices.

Appendices

Appendix A

Default Model Architectures

Default Ann Architectures utilized in thesis experimentations. Cifar10 Architecture is omitted from the appendix as it utilizes VGG16 logic delving too deep to showcase.

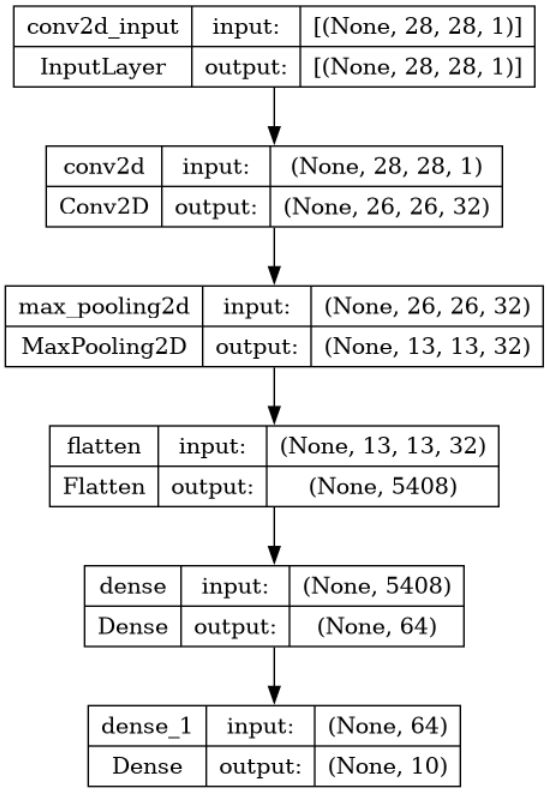


Figure A.1. *MNIST Default ANN*

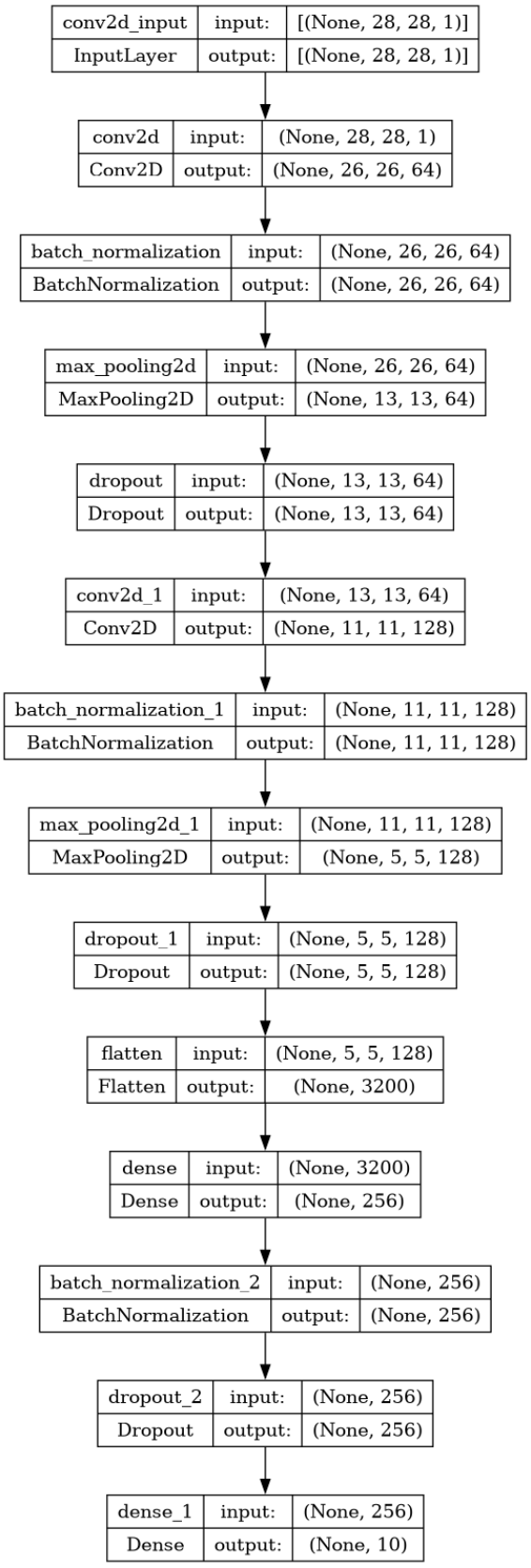


Figure A.2. *FashionMNIST Default ANN*

Bibliography

[1] Matt Zwolenski και Lee Weatherill. *The Digital Universe: Rich Data and the Increasing Value of the Internet of Things*. *Journal of Telecommunications and the Digital Economy*, 2:9, 2020.

[2] Ling Liu, Pan Zhou, Gang Sun, Xi Chen, Tao Wu, Hongfang Yu και Mohsen Guizani. *Topologies in Distributed Machine Learning: Comprehensive Survey, Recommendations and Future Directions*. *Neurocomputing*, 567:127009, 2024.

[3] Jakub Konečný, H. Brendan McMahan, Daniel Ramage και Peter Richtárik. *Federated Optimization: Distributed Machine Learning for On-Device Intelligence*, 2016.

[4] Junfei Qiu, Qihui Wu, Guoru Ding, Yuhua Xu και Shuo Feng. *A Survey of Machine Learning for Big Data Processing*. *EURASIP Journal on Advances in Signal Processing*, 2016(1):67, 2016.

[5] *The General Data Protection Regulation*. https://www.consilium.europa.eu/en/policies/data-protection/data-protection-regulation/.

[6] *California Consumer Privacy Act (CCPA) | State of California - Department of Justice - Office of the Attorney General*. https://oag.ca.gov/privacy/ccpa.

[7] Lina Zhou, Shimei Pan, Jianwu Wang και Athanasios V. Vasilakos. *Machine Learning on Big Data: Opportunities and Challenges*. *Neurocomputing*, 237:350–361, 2017.

[8] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson και Blaise Agueray Arcas. *Communication-Efficient Learning of Deep Networks from Decentralized Data*. *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, σελίδες 1273–1282. PMLR, 2017.

[9] Dinh C. Nguyen, Ming Ding, Pubudu N. Pathirana, Aruna Seneviratne, Jun Li και H. Vincent Poor. *Federated Learning for Internet of Things: A Comprehensive Survey*. *IEEE Communications Surveys & Tutorials*, 23(3):1622–1658, 2021.

[10] Ekaba Bisong. *Batch vs. Online Learning*. *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*Ekaba Bisong, επιμελητής, σελίδες 199–201. Apress, Berkeley, CA, 2019.

[11] Zheng Xu, Yanxiang Zhang, Galen Andrew, Christopher A. Choquette-Choo, Peter Kairouz, H. Brendan McMahan, Jesse Rosenstock και Yuanbo Zhang. *Federated Learning of Gboard Language Models with Differential Privacy*, 2023.

[12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le και Andrew Ng. *Large Scale Distributed Deep Networks*. *Advances in Neural Information Processing Systems*, τόμος 25. Curran Associates, Inc., 2012.

[13] Qiang Yang, Yang Liu, Tianjian Chen και Yongxin Tong. *Federated Machine Learning: Concept and Applications*, 2019.

[14] Tian Li, Anit Kumar Sahu, Ameet Talwalkar και Virginia Smith. *Federated Learning: Challenges, Methods, and Future Directions*. *IEEE Signal Processing Magazine*, 37(3):50–60, 2020.

[15] Mingzhe Chen, Deniz Gündüz, Kaibin Huang, Walid Saad, Mehdi Bennis, Aneta Vulgarakis Feljan και H. Vincent Poor. *Distributed Learning in Wireless Networks: Recent Progress and Future Challenges*. *IEEE Journal on Selected Areas in Communications*, 39(12):3579–3605, 2021.

[16] Konidaris Vissarion. *Extreme-Scale Online Machine Learning OnStream Processing Platforms*. 2022.

[17] Izchak Sharfman, Assaf Schuster και Daniel Keren. *A Geometric Approach to Monitoring Threshold Functions over Distributed Data Streams*. *Ubiquitous Knowledge Discovery: Challenges, Techniques, Applications*Michael May και Lorenza Saitta, επιμελητές, σελίδες 163–186. Springer, Berlin, Heidelberg, 2010.

[18] Vasileios Samoladas και Minos Garofalakis. *Functional Geometric Monitoring for Distributed Streams*. OpenProceedings.org, 2019.

[19] Niklas Kühl, Marc Goutier, Robin Hirt και Gerhard Satzger. *Machine Learning in Artificial Intelligence: Towards a Common Understanding*, 2020.

[20] *Artificial Intelligence : A Modern Approach*. https://thuvienso.hoasen.edu.vn/handle/123456789/8967.

[21] Haochen Hua, Yutong Li, Tonghe Wang, Nanqing Dong, Wei Li και Junwei Cao. *Edge Computing with Artificial Intelligence: A Machine Learning Perspective*. *ACM Computing Surveys*, 55(9):184:1–184:35, 2023.

[22] Amit Kumar Tyagi και Rekha G. *Machine Learning with Big Data*, 2019.

[23] Iqbal H. Sarker. *Machine Learning: Algorithms, Real-World Applications and Research Directions*. *SN Computer Science*, 2(3):160, 2021.

[24] Trevor Hastie, Jerome Friedman και Robert Tibshirani. *The Elements of Statistical Learning*, τόμος 9. Springer Science & Business Media, 2017.

[25] Thrasyvoulos Spyropoulos. *Reinforcement Learning*. *Class at the Department of Electrical and Computer Engineering , Technical University of Crete*, 2023.

[26] Yoshua Bengio, Aaron Courville και Pascal Vincent. *Representation Learning: A Review and New Perspectives*. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013.

[27] Vladimir Nasteski. *An Overview of the Supervised Machine Learning Methods*. *HORIZONS.B*, 4:51–62, 2017.

[28] Zoubin Ghahramani. *Unsupervised Learning*. *Advanced Lectures on Machine Learning: ML Summer Schools 2003, Canberra, Australia, February 2 - 14, 2003, Tübingen, Germany, August 4 - 16, 2003, Revised Lectures*Olivier Bousquet, Ulrike von Luxburg και Gunnar Rätsch, επιμελητές, σελίδες 72–112. Springer, Berlin, Heidelberg, 2004.

[29] Ofer Dekel. *From Online to Batch Learning with Cutoff-Averaging*. *Advances in Neural Information Processing Systems*, τόμος 21. Curran Associates, Inc., 2008.

[30] Diego Perez. *What Is Model Performance Measurement in Machine Learning?*, 2022.

[31] Haotian Zhang, Lin Zhang και Yuan Jiang. *Overfitting and Underfitting Analysis for Deep Learning Based End-to-end Communication Systems*. *2019 11th International Conference on Wireless Communications and Signal Processing (WCSP)*, σελίδες 1–6, 2019.

[32] Diego Perez. *Model Complexity in Machine Learning*, 2022.

[33] Machine Learning in Plain English. *Lesson 15 — Machine Learning: Overfitting, Underfitting, and Model Complexity Intuition*, 2023.

[34] A D Dongare, R R Kharde και Amit D Kachare. *Introduction to Artificial Neural Network*. 2(1), 2012.

[35] Yu chen Wu και Jun wen Feng. *Development and Application of Artificial Neural Network*. *Wireless Personal Communications*, 102(2):1645–1656, 2018.

[36] F. Rosenblatt. *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain*. *Psychological Review*, 65(6):386–408, 1958.

[37] Zhihua Zhang. *Artificial Neural Network*. *Multivariate Time Series Analysis in Climate and Environmental Research*Zhihua Zhang, επιμελητής, σελίδες 1–35. Springer International Publishing, Cham, 2018.

[38] Marius Constantin Popescu, Valentina E Balas, Liliana Perescu-Popescu και Nikos Mastorakis. *Multilayer Perceptron and Neural Networks*. 8(7), 2009.

[39] Marvin Minsky και Seymour A. Papert. *Perceptrons, Reissue of the 1988 Expanded Edition with a New Foreword by Léon Bottou: An Introduction to Computational Geometry*. MIT Press, 2017.

[40] *The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions | International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*. https://www.worldscientific.com/doi/abs/10.1142/s0218488598000094.

[41] Shiv Ram Dubey, Satish Kumar Singh και Bidyut Baran Chaudhuri. *Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark*. *Neurocomputing*, 503:92–108, 2022.

[42] Siddharth Sharma, Simone Sharma και Anidhya Athaiya. *ACTIVATION FUNCTIONS IN NEURAL NETWORKS*. *International Journal of Engineering Applied Sciences and Technology*, 04(12):310–316, 2020.

[43] Yasara Ransisi. *Neural Network Types.*, 2021.

[44] Keiron O'Shea και Ryan Nash. *An Introduction to Convolutional Neural Networks*, 2015.

[45] ODSC Community. *Understanding the Mechanism and Types of Recurrent Neural Networks*, 2020.

[46] Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus και Shimon Schocken. *Multilayer Feedforward Networks with a Nonpolynomial Activation Function Can Approximate Any Function*. *Neural Networks*, 6(6):861–867, 1993.

[47] M. Ghiassi, H. Saidane και D. K. Zimbra. *A Dynamic Artificial Neural Network Model for Forecasting Time Series Events*. *International Journal of Forecasting*, 21(2):341–362, 2005.

[48] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai και Tsuhan Chen. *Recent Advances in Convolutional Neural Networks*. *Pattern Recognition*, 77:354–377, 2018.

[49] Sumit Saha. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 Way*. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53, 2022.

[50] Petrakos Emmanouel. *Reconfigurable Logic (FPGA)-Based System Architecture for the Acceleration of Federated Learning in Neural Networks*. 2023.

[51] Cyber Marty. *Neural Networks Learning Process Explained*, 2023.

[52] Ian Goodfellow, Yoshua Bengio και Aaron Courville. *Deep Learning*. MIT Press, 2016.

[53] Meenal V. Narkhede, Prashant P. Bartakke και Mukul S. Sutaone. *A Review on Weight Initialization Strategies for Neural Networks*. *Artificial Intelligence Review*, 55(1):291–322, 2022.

[54] Michael Nielsen. *Neural Networks and Deep Learning*.

[55] Mirza Cilimkovic. *Neural Networks and Back Propagation Algorithm*.

[56] Sebastian Ruder. *An Overview of Gradient Descent Optimization Algorithms*, 2017.

[57] Saad Hikmat Haji και Adnan Mohsin Abdulazeez. *COMPARISON OF OPTIMIZATION TECHNIQUES BASED ON GRADIENT DESCENT ALGORITHM: A REVIEW*. *PalArch's Journal of Archaeology of Egypt / Egyptology*, 18(4):2715–2743, 2021.

[58] Atulanand. *GRADIENT DESCENT*, 2022.

[59] *Momentum and Learning Rate Adaptation*. https://willamette.edu/~gorr/classes/cs449/momrate.html.

[60] Yann LeCun, Leon Bottou, Genevieve B. Orr και Klaus Robert Müller. *Efficient BackProp*. *Neural Networks: Tricks of the Trade*Genevieve B. Orr και Klaus Robert Müller, επιμελητές, σελίδες 9–50. Springer, Berlin, Heidelberg, 1998.

[61] Diederik P. Kingma και Jimmy Ba. *Adam: A Method for Stochastic Optimization*, 2017.

[62] John Duchi, Elad Hazan και Yoram Singer. *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*.

[63] *Lecture 6.5-rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude | CiNii Research*. https://cir.nii.ac.jp/crid/1370017282431050757.

[64] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg και Tim Verbelen. *A Survey on Distributed Machine Learning | ACM Computing Surveys*. https://dl.acm.org/doi/abs/10.1145/3377454.

[65] Alexander Sergeev και Mike Del Balso. *Horovod: Fast and Easy Distributed Deep Learning in TensorFlow*, 2018.

[66] *O'Reilly Network: Distributed Systems Topologies: Part 2*.

[67] Surat Teerapittayanon, Bradley McDanel και H.T. Kung. *Distributed Deep Neural Networks Over the Cloud, the Edge and End Devices*. *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, σελίδες 328–339, Atlanta, GA, USA, 2017. IEEE.

[68] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio και Rafal Jozefowicz. *Revisiting Distributed Synchronous SGD*, 2017.

[69] *Asynchronous Decentralized Parallel Stochastic Gradient Descent*. https://proceedings.mlr.press/v80/lian18a.html.

[70] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson και Blaise Agüeray Arcas. *Communication-Efficient Learning of Deep Networks from Decentralized Data*, 2023.

[71] Cloud Hacks. *Federated Learning: A Paradigm Shift in Data Privacy and Model Training*, 2024.

[72] Loredana Caruccio, Gaetano Cimino, Vincenzo Deufemia, Gianpaolo Iuliano και Roberto Stanzione. *Surveying Federated Learning Approaches through a Multi-Criteria Categorization*. *Multimedia Tools and Applications*, 83(12):36921–36951, 2024.

[73] Yue Zhao, Meng Li, Liangzhen Lai, Naveen Suda, Damon Civin και Vikas Chandra. *Federated Learning with Non-IID Data*. 2018.

[74] Rajarshi Bhose. *Differentiating between Distributed Learning and Federated Learning*, 2023.

[75] Omair Rashed Abdulwareth Almanifi, Chee Onn Chow, Mau Luen Tham, Joon Huang Chuah και Jeevan Kanesan. *Communication and Computation Efficiency in Federated Learning: A Survey*. *Internet of Things*, 22:100742, 2023.

[76] Myeongsuk Pak και Sanghoon Kim. *A Review of Deep Learning in Image Recognition*.

[77] Y. Lecun, L. Bottou, Y. Bengio και P. Haffner. *Gradient-Based Learning Applied to Document Recognition*. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[78] Karen Simonyan και Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2015.

[79] Kaiming He, Xiangyu Zhang, Shaoqing Ren και Jian Sun. *Deep Residual Learning for Image Recognition*, 2015.

[80] Minos Garofalakis, Daniel Keren και Vasilis Samoladas. *Sketch-Based Geometric Monitoring of Distributed Stream Queries*. *Proceedings of the VLDB Endowment*, 6(10):937–948, 2013.

[81] Alon Noga, Matias Yossi και Szegedy Mario. *The Space Complexity of Approximating the Frequency Moments*. 1996.

[82] Theologitis Michael. *Algorithms for Online Federated Machine Learning*. 2023.

[83] Frangias Georgios. *Federated Learning at TensorFlow Using the Geometric Approach*. 2023.

[84] Emily Ekaireb, Xiaofan Yu, Kazim Ergun, Quanling Zhao, Kai Lee, Muhammad Huzaifa και Tajana Rosing. *Ns3-Fl: Simulating Federated Learning with Ns-3*. *Proceedings of the 2022 Workshop on Ns-3*, σελίδες 97–104, Virtual Event USA, 2022. ACM.

[85] *Modelling Heterogeneity With and Without the Dirichlet Process - Green - 2001 - Scandinavian Journal of Statistics - Wiley Online Library*. https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-9469.00242.

[86] Daniel J. Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Javier Fernandez-Marques, Yan Gao, Lorenzo Sani, Kwing Hei Li, Titouan Parcollet, Pedro Porto Buarque de Gusmão και Nicholas D. Lane. *Flower: A Friendly Federated Learning Research Framework*, 2022.

[87] *Electronics | Free Full-Text | Computer Network Simulation with Ns-3: A Systematic Literature Review*. https://www.mdpi.com/2079-9292/9/2/272.