POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Tesi di Laurea Magistrale in
Computer Science and Engineering
Ingegneria Informatica

Dynamic Selection Techniques for Federated Learning

Author:
Andrea Restelli

Student ID:
995512

Advisor:
Prof. Luciano Baresi

Co-advisor:
Tommaso Dolci

Academic Year:
2022-23

Dedicated to my family.

Abstract

The rise of machine learning (ML), fueled by open datasets, affordable processing, and cost-effective storage, has accelerated model development across various applications like computer vision and natural language processing. Federated learning (FL) enables the creation of ML models without compromising user data privacy. Users train individual models and periodically share updates with a central server. The server aggregates these models, creating a centralized model that incorporates insights from all users without accessing their specific data. This approach maintains data privacy while deriving collective knowledge from the distributed user data. FL introduces complexities such as statistical and system heterogeneity, requiring innovative algorithms for convergence in non-iid datasets. Furthermore, strategic client selection and resource allocation become crucial for optimizing FL system performance. Finally, experimenting with realistic federated environments is challenging due to associated costs.

This thesis proposes extensions to Flower, a highly promising framework for advancing federated learning research. Using Flower to prioritize reproducibility and extendability, crucial for experimental challenges in FL, the thesis extend it with algorithms for dynamic client selection and resource-aware workload allocation.

The work concludes with an experimental phase assessing the impact of introduced elements compared to state-of-the-art techniques. Experimental results showcase the competitiveness of proposed strategies, particularly in heterogeneous settings, demonstrating effectiveness, convergence speed, and stability. The experiments underscore the importance of strategic client selection and workload distribution in FL for effective and stable model training. The thesis contributes to advancing FL research and highlights Flower as a valuable framework for future development.

Keywords: machine learning, federated learning, framework, dynamic selection, dynamic workload allocation

Abstract in lingua italiana

L'ascesa del machine learning (ML), alimentata da set di dati disponibili, elaborazione a prezzi accessibili e archiviazione a costi contenuti, ha accelerato lo sviluppo di modelli in varie applicazioni come la computer vision e l'elaborazione del linguaggio naturale. Il federated learning (FL) consente la creazione di un modello di apprendimento automatico senza compromettere la privacy dei dati degli utenti. Gli utenti addestrano modelli individuali e condividono periodicamente gli aggiornamenti con un server centrale. Il server aggrega questi modelli, creando un modello centralizzato che incorpora gli aggiornamenti di tutti gli utenti senza accedere ai loro dati specifici. Questo approccio consente di mantenere la privacy dei dati e di ricavare conoscenze collettive dai dati distribuiti degli utenti. Il FL introduce complessità come l'eterogeneità statistica e del sistema, che richiede algoritmi innovativi per la convergenza in insiemi di dati non indipendenti e identicamente distribuiti. Inoltre, la selezione strategica dei client e l'allocazione delle risorse diventano cruciali per ottimizzare le prestazioni del sistema FL. Infine, la sperimentazione di ambienti federati realistici è impegnativa a causa dei costi associati.

Questa tesi propone Flower come un framework molto promettente per far progredire la ricerca sul federated learning. Utilizzando Flower per dare priorità alla riproducibilità e all'estendibilità, cruciali per le sfide sperimentali nel federated learning, la tesi lo estende con algoritmi per la selezione dinamica dei client e l'allocazione del carico di lavoro consapevole delle risorse.

Il lavoro si conclude con una fase sperimentale che valuta l'impatto degli elementi introdotti rispetto allo stato dell'arte. I risultati sperimentali dimostrano la competitività delle strategie proposte, in particolare in contesti eterogenei, dimostrando efficacia, velocità di convergenza e stabilità. Gli esperimenti sottolineano l'importanza della selezione strategica dei client e della distribuzione del carico di lavoro nel federated learning per una formazione efficace e stabile dei modelli. La tesi contribuisce a far progredire la ricerca sul federated learning e mette in evidenza come Flower sia un valido framework per lo sviluppo futuro della materia.

Parole chiave: machine learning, federated learning, framework, selezione dinamica,

allocazione dinamica del carico computazionale

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

1 Introduction **1**

1.1 Problem statement . . . 2
1.2 Contribution of the thesis . . . 3
1.3 Structure of the thesis . . . 4

2 Machine learning **5**

2.1 Basics . . . 5
2.1.1 Approaches . . . 5
2.1.2 Training . . . 6
2.1.3 Inference . . . 7
2.2 Deep Learning . . . 8
2.2.1 Multi layer perceptrons . . . 10
2.2.2 Convolutional Neural Networks . . . 11
2.3 Paradigms . . . 13
2.3.1 Centralized ML . . . 13
2.3.2 Distributed ML . . . 13

3 Federated learning **15**

3.1 Federated Learning . . . 15
3.1.1 Federated learning algorithms . . . 16
3.1.2 Open Issues . . . 21
3.2 Analysis of FL frameworks . . . 22
3.2.1 Tensorflow federated . . . 23

3.2.2 FATE . . . 24
3.2.3 FedJax . . . 25
3.2.4 FedML . . . 26
3.2.5 PySyft . . . 26
3.2.6 Flower . . . 27
3.3 Framework of choice . . . 28
3.3.1 Criteria . . . 28
3.3.2 Flower in detail . . . 30

4 Implementation **37**
4.1 Dynamic selection . . . 37
4.1.1 FedAvg . . . 38
4.1.2 Dynamic sampling . . . 40
4.1.3 Power of choice . . . 42
4.2 Resource-aware workload selection . . . 47
4.2.1 Static optimizer . . . 49
4.2.2 Uniform optimizer . . . 50
4.2.3 RT optimizer . . . 51
4.2.4 Equal computation time optimizer . . . 52

5 Experiments **57**
5.1 Setup . . . 57
5.1.1 Task . . . 57
5.1.2 Dataset loading . . . 58
5.1.3 Simulation parameters . . . 60
5.2 Results . . . 61
5.2.1 Dynamic selection . . . 61
5.2.2 Resource-aware Workload Selection . . . 68
5.2.3 Takeaways from the experiments . . . 72

6 Conclusions **75**
6.1 Future work . . . 76

Bibliography **79**

List of Figures **87**

List of Tables 89

Acknowledgements 91

1 | Introduction

Machine Learning (ML) is a specialized domain within the broader field of Artificial Intelligence (AI) and computer science, focused on employing data and algorithms to replicate human learning processes. While AI encompasses a wide array of technologies emulating human capabilities, ML specifically trains machines in the art of learning. The increasing prevalence of ML applications is primarily attributable to the proliferation of diverse and extensive datasets, cost-effective and potent computational processing, as well as affordable data storage solutions. These factors collectively enable the rapid and automated development of models capable of handling larger and more intricate datasets, thereby yielding quicker and more precise outcomes.

The applications of ML span a multitude of domains. Notably, it finds extensive application in computer vision, where it empowers computers and systems to derive meaningful insights from digital images, videos, and other visual inputs, thereby facilitating informed decision-making. Such applications extend to recommendation systems in e-commerce and content platforms, radiology imaging within the healthcare sector, and the realization of self-driving cars in the automotive industry.

The prominence of deep learning within the domain of machine learning has led to a significant dependence on the accumulation of extensive datasets for model development. However, this increased demand for data has given rise to notable challenges in terms of data transmission and storage. Specifically, the transfer of substantial data samples from external devices can lead to network latency issues, and concurrently, there are concerns regarding privacy due to the potential inclusion of sensitive information within the collected datasets.

Federated Machine Learning (Federated Learning or FL) offers a partial solution to these challenges. FL constitutes an ML technique that facilitates the training of algorithms across a network of decentralized devices, each housing local data samples. Crucially, in FL, local data samples remain within their respective devices and are never exchanged between devices. The training process is performed locally by a subset of devices. Participating devices retrieve the model from a central server, conduct local model training

with their respective data, and subsequently communicate the resultant model back to the server. These model weights are then aggregated, and the process can iteratively repeat, referred to as "rounds," until a satisfactory level of accuracy is attained.

Numerous tools support federated learning model training, yet no standard has emerged. The dynamic nature of this recent paradigm is reflected in the evolving landscape of tools. Additionally, there are gaps in the analysis of state-of-the-art techniques and tools, indicating opportunities for further exploration and refinement in this rapidly evolving domain.

1.1. Problem statement

Federated Learning (FL) offers distinct advantages over traditional Machine Learning (ML) in distributed settings. FL enhances privacy by allowing local model training on decentralized devices, reducing the need for data sharing and minimizing privacy risks. Additionally, it reduces communication overhead as data remains on the local device, alleviating network congestion.

However, FL introduces its own set of complexities. In contrast to conventional ML, which conducts training centrally on a single server, Federated Learning (FL) adopts a decentralized approach with parallel training across multiple devices. This decentralization introduces new challenges that need innovative solutions.

One of the distinguishing feature between classic ML and FL is the presence of statistical and system heterogeneity. Statistical heterogeneity arises from the non-iid (non-independent and identically distributed) nature of data on different devices, coupled with significant variations in the sizes of local data samples, influenced by individual user behaviors. This diversity in data distribution can impede model convergence compared to the homogeneous centralized dataset used in traditional ML. To address this, techniques and algorithms are required to bridge the gap between locally obtained models and ensure convergence.

System heterogeneity encompasses variations in device attributes such as computational capabilities, memory, energy availability, and environmental factors like network speed and reliability. In contrast to the relatively uniform and robust machines used in classic ML, FL contends with diverse and less predictable system conditions.

To address these challenges, specialized platforms and algorithms are imperative. Strategic client selection and resource allocation become crucial in optimizing FL system performance. Considering factors such as quality and size of local data, computational resources,

and device availability can enhance client selection, mitigating the impact of slower clients and refining the learning process. These strategies must be tailored to accommodate the diverse and unpredictable conditions in FL, thereby optimizing both model accuracy and resource utilization.

Existing solutions have tried to address these challenges by means either of static solutions that don't take into account the evolution of the learning over the duration of the process or either solutions that don't take into account system heterogeneity.

Lastly, experimentation is difficult, given the fact that setting up a realistic federated network for experimentation presents notable challenges and costs. A real federated environment often includes an extensive number of clients. Replicating such a vast and diverse environment mandates acquiring or renting a multitude of devices, resulting in substantial costs.

1.2. Contribution of the thesis

The goal of this thesis is to address the challenges outlined in Section 1.1, exploring methodologies and innovations to maximize the potential of Federated Learning while tackling its inherent complexities.

This thesis has three primary objectives: (1) analyze available Federated Learning (FL) frameworks and select the most promising one, (2) extend it with algorithms for dynamic client selection during training, and (3) further extend it with algorithms enabling resource-aware workload allocation to clients during training.

A significant contribution of this thesis is the use of an open-source framework, prioritizing reproducibility and extendability. As highlighted in Section 1.1, experimentation poses a critical challenge in federated learning. Thus, implementing the solution in an environment favorable to easy experiment reproduction in a simulated federated setting is crucial. At the same time, to obtain meaningful results, the experimental environment should allow reproducing conditions close to the real ones.

The work concludes with an experimental phase to assess the impact of the introduced elements compared to state-of-the-art techniques. This experimental phase aims to closely replicate a real federated environment and is designed for ease of future reproducibility and extension.

1.3. Structure of the thesis

The thesis is organized as follows. In Chapter 2, we introduce the theoretical foundations on which this thesis is based, specifically focusing on the topic of machine learning. Chapter 3 presents the concept of federated learning, the paradigm of machine learning upon which this thesis is focused. In this chapter, we also analyze available federated learning frameworks, highlighting their strengths and weaknesses. Chapter 4 delves into the implementation of the proposed techniques to address challenges introduced in Section 1.1. The techniques primarily revolve around two key topics: dynamic selection of clients and resource-aware workload allocation. In Chapter 5, we present the experiments conducted to assess the impact of the introduced strategies. This chapter not only focuses on the results obtained but also describes the setup in detail to facilitate the reproducibility of the experiments. Lastly, in Chapter 6, we summarize the contributions of the thesis and examine possible future work.

2 | Machine learning

This chapter describes the theoretical foundations on which this thesis is based. Section 2.1 gives an introduction of what machine learning is. Section 2.2 dives into what is deep learning and what distinguishes it from traditional ML.

2.1. Basics

Machine Learning (ML) is a branch of computational research and algorithmic techniques with a primary focus on identifying inherent patterns within data and employing these patterns to construct models able to predict the correct output given a certain input.

Performing machine learning involves creating a model which is trained on some training data and then can process additional data to make predictions. Various types of models have been used and researched for machine learning systems: decision trees, support-vector machines, regression analyses, bayesian networks and artificial neural networks.

2.1.1. Approaches

Machine learning approaches can be broadly categorized into three main paradigms based on the type of feedback or signal available to the system:

1. **Supervised Learning:** This paradigm learns a model by mapping an input to an output using examples of input-output pairs, known as experience. Human-labeled data is essential as it allows comparison of the model's predicted output with the actual output decided by human experts. For instance, in image classification, a model is trained on a dataset of images labeled with their respective classes. Given a new image, the trained classifier predicts the corresponding class.
2. **Unsupervised Learning:** This approach identifies undetected patterns or regularities in the data to create a representation used for probabilistic reasoning or prediction. As there are no predefined outputs in the training data, it does not rely on human-labeled examples. Clustering serves as an example of this paradigm.

Here, the algorithm analyzes data to detect similarities (features) and, based on these features, groups the data. The number and type of groups are not predefined.

3. **Reinforcement Learning:** In this paradigm, a system operates in an environment where it can perform various actions. Each action influences the environment and is associated with a reward. The objective is to select actions that maximize a reward function over the long term.

This thesis will be focused only on supervised learning. The workflow in supervised learning generally unfolds through two main stages: the training phase and the inference phase, each with its own specific goals and techniques.

2.1.2. Training

The training phase involves learning (determining) good values for all the weights and the bias from labeled examples. In supervised learning, a machine learning algorithm builds a model by examining many examples and attempting to find a model that minimizes loss; this process is called *empirical risk minimization* [18].

Loss function

A cardinal element of the learning process involves employing a loss function. This function serves a fundamental purpose: it measures how much the model's predictions ($\hat{y}$) deviate from the actual outcomes (y). There are different types of loss functions, depending on the nature of the task.

In regression tasks, the Mean Squared Error (MSE) [2] [55] is often used as the loss function, and is described mathematically as:

$$L(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Here, n represents the total number of observations. The function provides a singular scalar value, representing the mean of the squared differences between predicted and actual outcomes, offering a measurable insight into the precision of the model's predictions.

In classification problems the Cross-Entropy loss function is predominantly used [53]. Consider p_i to be the probability, as computed by the model, asserting that a particular sample belongs to class i. Given the truth values are one-hot encoded – for instance, [1,0,0], [0,1,0], and [0,0,1] – and asserting that t_i represents the truth value correlative to class i, the value of t_i is discerned to be 1 if the sample is a member of class i and 0

otherwise.

The Cross-Entropy loss function quantifies the distance between output probabilities p and actual truth values. For a singular data point, it is expressed as follows:

$$L(t, p) = -\sum_{i=1}^{N} t_i \log(p_i)$$

In scenarios where the actual labels are one-hot encoded, we refer to the methodology as "Categorical Cross-Entropy". On the contrary, when labels are expressed as unique integers, for example [1], [2], and [3] in a 3-class classification problem, we employ the term "Sparse Cross-Entropy".

After identifying the loss, the objective shifts to minimizing this computed error, usually by employing optimization algorithms. A widely utilized algorithm in this context is Gradient Descent [56] [7]. This method operates by repetitively altering the model parameters (θ) in a way that continuously reduces the computed loss. The repeated adjustment or update to the model parameters can be mathematically represented as:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta L(y, \hat{y})$$

In this representation:

- θ_{t+1} and θ_t symbolize the parameter values at the subsequent and current iteration, respectively,
- α denotes the learning rate, dictating the magnitude of steps taken towards minimizing the loss function, and
- $\nabla_\theta L$ signifies the gradient of the loss function concerning the model parameters, pointing towards the steepest ascent of the loss function in the parameter space.

After concluding the training phase, the model transitions to the second stage, the inference phase.

2.1.3. Inference

In this phase the model, comprehensively trained using a training dataset, is employed to make predictions or decisions on new, unseen data, without undergoing further modifications.

Inference involves using the model f with learned parameters θ to predict outputs $\hat{y}$ given new inputs X, typically expressed as:

$$\hat{y} = f(X; \theta)$$

Here, θ remains constant, and the model applies the patterns recognized during training to new data.

The model's performance in this phase is evaluated using specific metrics, chosen based on the type of ML problem. For instance, precision and recall may be employed for classification problems, while mean absolute error (MAE) might be utilized for regression problems to average out the absolute differences between the predicted and actual values.

The inference phase is crucial for testing the model's ability to generalize its learned knowledge to new, unobserved data, ensuring its predictions or decisions are robust and reliable across diverse real-world applications.

2.2. Deep Learning

Deep Learning (DL) is a specialized branch of Machine Learning (ML) that stands out for its use of deep neural networks [16]. Unlike traditional ML methods, which often require experts to manually design features to understand data, DL relies on complex neural structures with multiple layers of artificial neurons. These layers equip DL models with the remarkable ability to independently and hierarchically recognize complex data patterns directly from raw input.

To simplify, a deep neural network can be thought as a series of interconnected computational units, similar to neurons in the human brain.

Each neuron in a DNN performs a weighted sum of its input values and applies an activation function to produce an output. The weights associated with these connections are crucial and are iteratively adjusted during the training process to optimize the network's performance.

Mathematically, let us consider a single neuron in a DNN. It receives inputs, denoted as $x_1, x_2, x_3, \ldots, x_n$, each multiplied by corresponding weights $w_1, w_2, w_3, \ldots, w_n$. Additionally, there is a bias term, b, which plays an essential role in shifting the activation function. The weighted sum, also known as the activation, is calculated as:

$$\text{Activation} = w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3 + \ldots + w_n \cdot x_n + b$$

Afterward, an activation function (commonly a non-linear function like the sigmoid or ReLU) is applied to this result to introduce non-linearity into the neuron's response. The neuron's output, denoted as y, is then determined as:

$$y = \text{Activation_Function}(\text{Activation})$$

In Figure 2.1 the overall architecture of a neuron in an artificial neural network is shown (the bias term is omitted for simplicity).

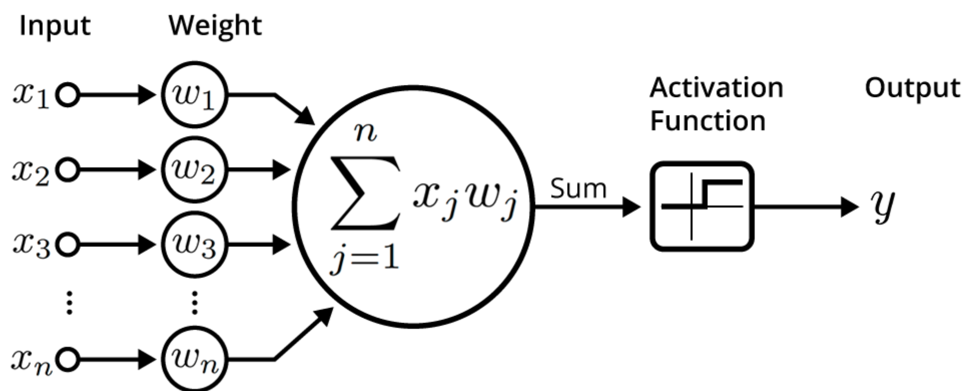

Figure 2.1: The architecture of a neuron in a DNN.

This process is repeated across multiple layers, with each layer learning progressively abstract features from the input data. The weights for each connection are fine-tuned using optimization algorithms like gradient descent during the training phase, where the network minimizes a loss function, such as Mean Squared Error or Cross-Entropy, to improve its predictive accuracy.

This iterative process of weighted summation, activation, and weight adjustment across multiple layers enables deep neural networks to autonomously discover intricate data representations and capture complex patterns, making them highly effective for tasks such as image recognition, natural language processing, and more.

The key appeal of deep learning lies in its automatic discovery of high-level data features, leading to significantly improved accuracy, especially in tasks like image recognition and speech analysis. DL's attractiveness is rooted in its efficiency in grasping complex data relationships while leveraging modern computing power and abundant datasets. Consequently, deep learning has swiftly integrated into various applications, including natural language understanding [42], computer vision [50], autonomous systems, and more. In the current landscape of machine learning, DL has emerged as an influential and prevalent

paradigm.

2.2.1. Multi layer perceptrons

A Multilayer Perceptron (MLP) is a feedforward neural network that comprises multiple layers of interconnected neurons [1]. This structure is mathematically described as follows:

Input Layer: The first layer is the input layer, consisting of neurons representing input features $(x_1, x_2, \ldots, x_n)$. Each neuron in this layer simply passes its value to the neurons in the subsequent layer.

Hidden Layers: Between the input and output layers, there are hidden layers where the network learns and captures complex patterns. Neurons in the hidden layers are represented mathematically as:

$$\text{Output} = \text{Activation}_i \left(\sum_{j=1}^{m} w_{ij} \cdot x_j \right)$$

Here, Activation_i is the activation function for neuron i, x_j is the output of the j-th neuron from the previous layer, w_{ij} is the weight for the connection between neuron j and neuron i, and the summation covers all neurons in the previous layer.

Output Layer: The last layer is the output layer, producing the final result or prediction. Neurons in the output layer are represented as:

$$\text{Output} = \text{Activation}_k \left(\sum_{i=1}^{p} w_{ik} \cdot x_i \right)$$

Here, Activation_k is the activation function for neuron k, x_i is the output of the i-th neuron from the last hidden layer, w_{ik} is the weight for the connection between neuron i and neuron k, and the summation covers all neurons in the last hidden layer.

During training, weights (w_{ij} and w_{ik}) are adjusted using optimization algorithms, like gradient descent, to minimize a loss function. The goal is to learn optimal weights that enable the network to make accurate predictions.

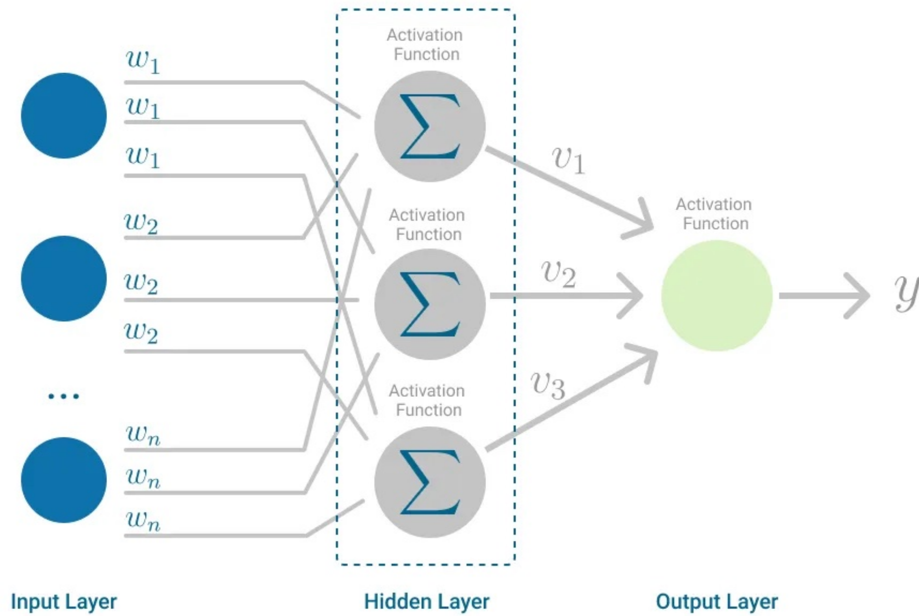

Figure 2.2: A MLP with one output neuron.

MLPs excel at capturing intricate patterns and relationships within structured data, making them suitable for a diverse array of applications, spanning classification and regression tasks.

2.2.2. Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [52] are a class of deep learning models specifically designed for processing grid-like data, such as images. CNNs utilize convolutional layers to automatically detect spatial patterns and hierarchical features in the input data. The core operation in a convolutional layer is the convolution operation, defined as:

$$\text{Output}(x, y) = \sum_i \sum_j \text{Input}(x - i, y - j) \cdot \text{Kernel}(i, j)$$

In this equation, $\text{Output}(x, y)$ represents the value at position (x, y) in the output feature map, $\text{Input}(x, y)$ is the value at position (x, y) in the input data, and $\text{Kernel}(i, j)$ is the value at position (i, j) in the convolutional kernel (filter).

After the Convolutional Neural Network (CNN) has processed the input data with convolutional layers, the resulting feature maps contain information about local patterns and features. However, these feature maps are not directly interpretable for classification tasks. To make a final prediction or classification, a fully connected layer is typically

added to the network.

The dense layer takes the flattened feature maps as input, where each element in the feature maps corresponds to a learned feature, and passes them through a series of fully connected neurons. These neurons collectively learn to recognize and combine the features from the feature maps, allowing the network to make high-level decisions about the input data. The output layer of the dense layer often consists of one or more neurons, depending on the specific classification task.

In classification tasks the output layer has as many neurons as there are classes or categories to classify the input image into. The network's final prediction is based on the activations of these output neurons, typically using a softmax activation function to convert raw scores into class probabilities. In Figure 2.3, a simple Convolutional Neural Network (CNN), taking an image from the MNIST dataset [54] as input, features 10 output neurons, each outputting the probability of the input image representing one of the 10 digits.

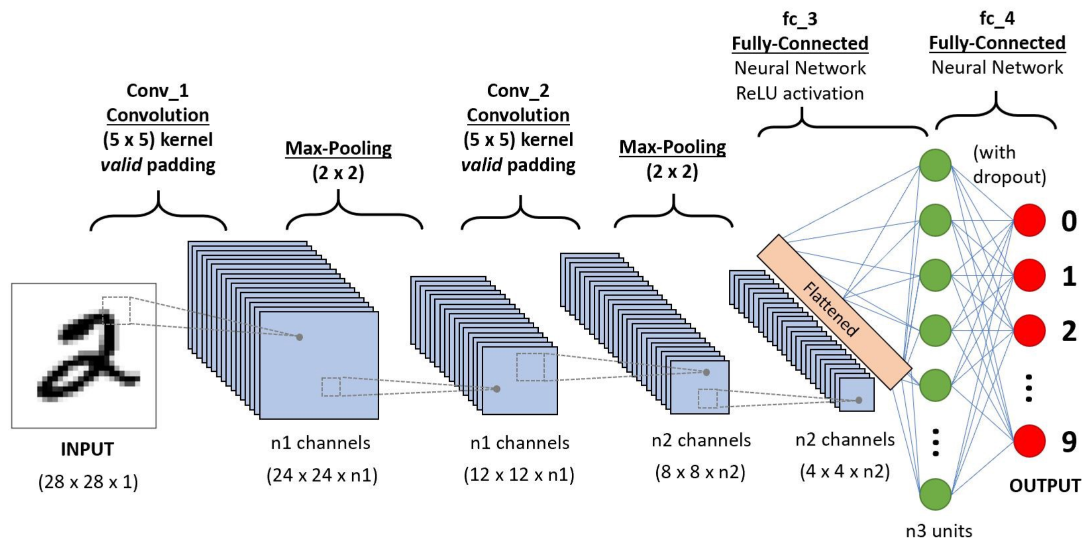

Figure 2.3: The overall architecture of a CNN.

The integration of convolutional layers for feature extraction and dense layers for classification makes CNNs well-suited for tasks like image recognition, object detection, and more. The hierarchical features extracted by the convolutional layers provide the dense network with a rich set of features to make informed classifications, and this combination has proven highly effective in various domains.

2.3. Paradigms

Together with the growing volume of data and the increasing computational capabilities, machine learning paradigms have evolved in the way they manage the training process.

2.3.1. Centralized ML

In the traditional paradigm of centralized learning (CL), client devices extract raw data from various sources spanning measurements, audio, images to videos, and, post preliminary pre-processing, relay it to a centralized server responsible for executing computationally intensive model training tasks. This methodology, while consolidating computational demands at a single point, introduces substantial traffic overheads to the underlying network due to the requisite transmission of voluminous data essential for training models.

Although this framework absolves participating devices from computationally taxing training processes, it introduces a dichotomy: devices might be hesitant to share data, especially of a privacy-sensitive nature, while the necessity to transmit extensive data imposes a palpable communication overhead between the participants and the central server, potentially hampering the efficacy and responsiveness of the learning process. Consequently, CL, while proficiently centralizing computational demands, at the same time introduces challenges pertaining to data privacy and communication overhead, necessitating careful consideration and mitigation.

2.3.2. Distributed ML

Driven by the growing concerns and extra communication efforts linked to centralizing data in Machine Learning (ML), new approaches that focus on sharing the workload and keeping data private gained interest [38]. The first paradigm, called distributed (on-site) learning, avoids sending user data and requests to the cloud, choosing instead to use on-site ML. Here, the server sends a ready-made or general ML model to user devices. After it is set up, each device tweaks the model by training it with its own data, allowing for personalized predictions and data analysis without giving away private data. This method, which clearly keeps data safe by holding it on the user's device, has been used in various applications, from detecting skin cancer [12] to smart classrooms [43].

However, without back-and-forth data exchange between the cloud and devices, models are limited to individual user experiences and do not benefit from other users' data.

Federated Learning (FL) provides a way to build a shared Machine Learning (ML) model

while letting users keep their data. In this approach, users each train their own models and send them periodically to a central server. The server combines all the models into one, benefiting from the knowledge of all users without needing to access their individual data. This method keeps private data spread out across users, while still creating a centralized model that reflects insights from the data of all the clients participating in the training.

Therefore, distributed ML introduces various strategies to mitigate the challenges posed by centralized learning.

Figure 2.4 illustrates a comparison among the three approaches.

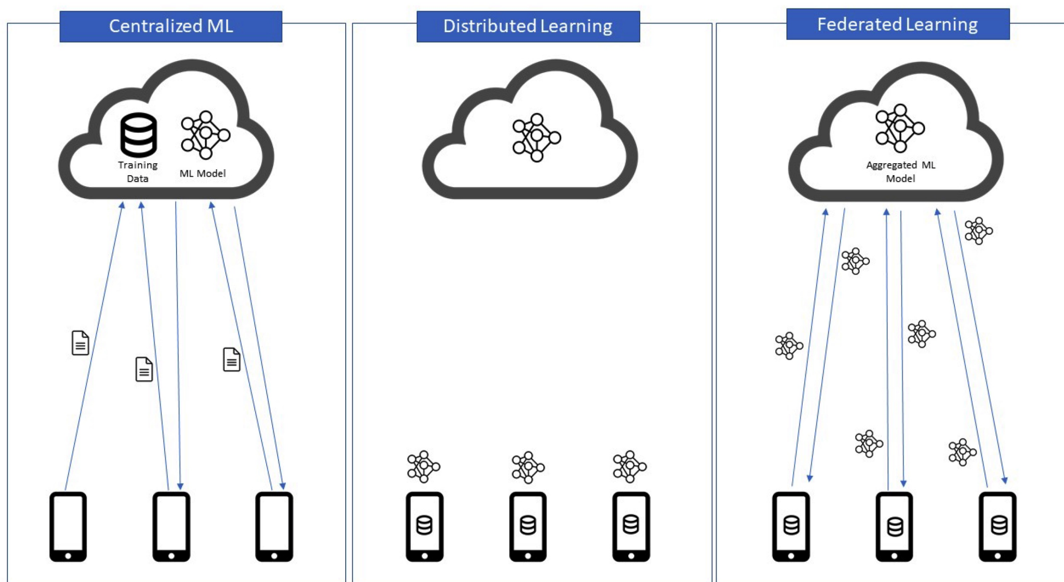

Figure 2.4: Comparison between the three approaches.

3 | Federated learning

This chapter introduces the concepts of Federated Learning this thesis is based on. Section 3.1 introduces the topic of Federated Learning. Section 3.2 analyses the available federated learning frameworks.

3.1. Federated Learning

Federated learning is a machine learning paradigm that trains an algorithm across multiple decentralized devices, each using its own dataset [30]. The first key point of federated learning is that it enables multiple actors to build a common, robust machine learning model without sharing data, thus addressing critical issues such as data privacy, data security, data access rights and access to heterogeneous data. This aspect enables federated learning to be used in scenarios where privacy is crucial, such as for example in training models with data coming from different hospitals, holding sensitive data from patients [44], or large language models to be trained on chats of users [22].

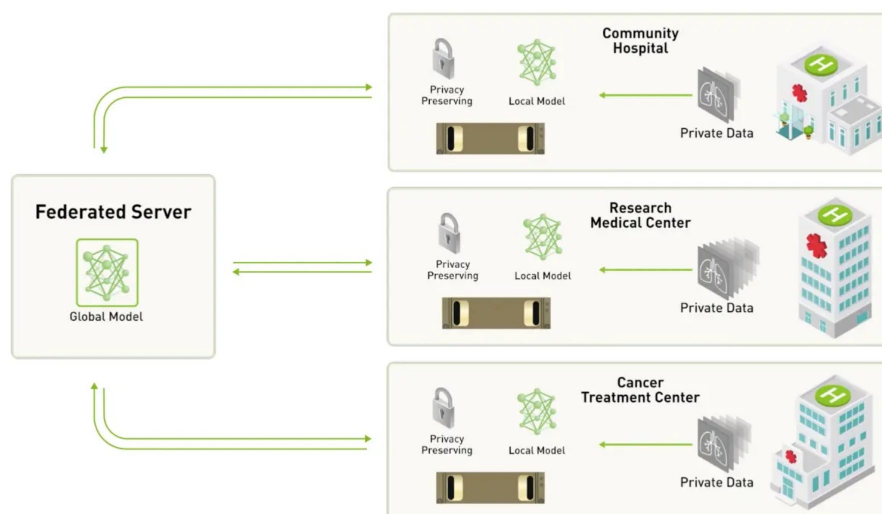

Figure 3.1: Federated learning applied to the healthcare sector.

Another key point of FL is that participating devices are highly heterogeneous in terms

of computing resources, data (non-IID), and network connections. Devices are equipped with different hardware and are located in dynamic and diverse environments. The participating devices are usually mobile or edge devices with limited resources. Each device is provided with a local dataset whose samples are usually generated by the interaction of the user with the device. For example, the local dataset can be composed of pictures taken with a mobile phone camera. In other cases the local dataset could contain the sentences written by the user with a virtual mobile keyboard [22].

The training phase is based on iterative training rounds. During a round, a subset of devices is chosen to participate in the training. These devices receive the current model weights from a central server, train the model using their own local dataset, and then relay back to the server the updated weights of the model. After receiving the weights of the model updated from the devices, the server aggregates them to a central model and proceeds to the following round. This iterative process ensures each device contributes with its local insights, leading to a model that embodies the collective wisdom of all participants, aiming for optimal accuracy and generalizability, while preserving the data privacy of the clients.

3.1.1. Federated learning algorithms

Modifications to the way the server behaves in orchestrating clients along rounds and aggregating weights are frequent, and they take the name of federated learning algorithms.

FedAvg

The most common algorithm, usually used as benchmarking baseline in research papers, is called FedAvg, introduced by McMahan et al. [40]. This algorithm combines local stochastic gradient descent (SGD) on each client with a server that performs model averaging.

Consider a setting with a predetermined set of K clients. Each client k has a unique local dataset of size n_k. At the beginning of every round, a random subset S_t of clients is chosen, which represents a fraction C of the total clients. The variable m denotes the count of selected clients for round t. Initially, it's assumed that all K clients are chosen, i.e., $C = 1$. Let n be the cumulative data size from all selected clients, given as $\sum_{k=1}^{K} n_k = n$, and let η be the learning rate.

The process starts with the server dispatching the global model parameters w_t to the selected clients. These clients then train this global model locally on their local dataset. Specifically, using the loss function F_k, every client k adjusts its local model via the

equation $w_k = w_k - \eta \nabla F_k(w_k)$, executing this update for E epochs.

Afterwards, the server aggregates these updates to determine the new global model, which is an average based on the size of the local datasets. The formula for the global weights at the completion of round t is:

$$w_{t+1} = w_t - \eta \sum_{k=1}^{K} \frac{n_k}{n} g_k \tag{3.1}$$

An alternative formulation of this update, considering the weights $w_{k,t}$ of each client k at time t is:

$$w_{k,t+1} = w_{k,t} - \eta g_k \tag{3.2}$$

Combining equations (3.1) and (3.2), we derive:

$$\begin{aligned} w_{t+1} &= w_t - \eta \sum_{k=1}^{K} \frac{n_k}{n} g_k \\ &= \sum_{k=1}^{K} \frac{n_k}{n} (w_{k,t} - \eta g_k) \\ &= \sum_{k=1}^{K} \frac{n_k}{n} w_{k,t+1} \end{aligned} \tag{3.3}$$

This aggregation strategy gives more importance to weights from clients who've used larger data subsets for local training. The underlying principle is that clients with more data typically yield better models, warranting a higher influence on the global model.

FedProx

The FedProx federated algorithm proposed by Li et al. [46] demonstrates significantly more stable and accurate convergence behavior relative to FedAvg in highly heterogeneous settings. When speaking about heterogeneity, both statistical and system heterogeneities are considered.

System Heterogeneity: System heterogeneity in federated networks refers to the differences in system-level attributes of devices. The computational, storage, and communication capacities can vary based on hardware (e.g., CPU, memory), network connectivity (3G, 4G, 5G, WiFi), and power constraints (battery level). In FedAvg, devices with constrained resources, which cannot compute E epochs within a set time frame, are excluded.

This can impact convergence by reducing the effective devices contributing to the training and potentially introducing bias if dropped devices possess specific data traits. FedProx addresses this by allowing the number of epochs on each device to vary according to its system resources.

Statistical Heterogeneity: Statistical heterogeneity arises when devices have non-uniform data distributions. As noted in [40], when data is non-identically distributed, the model can diverge empirically. To prevent this, FedProx introduces a proximal term to the local subproblem, thereby limiting the variability of local updates. This term serves dual purposes: (1) it counters statistical heterogeneity by constraining local updates to remain close to the initial global model; and (2) it allows for safely incorporating variable amounts of local work resulting from systems heterogeneity. More precisely, instead of just minimizing the local function $F_k(w)$, device k aims to minimize the following objective:

$$h_k(w; w_t) = F_k(w) + \frac{\mu}{2}\|w - w_t\|^2 \tag{3.4}$$

where $\mu > 0$ is a hyper-parameter and w_t represents the global model. Through experimentation, the authors determine that μ can be adaptively set based on model performance. A large μ can slow convergence by constraining updates to be near the starting point, while a small μ might have minimal impact. Their heuristic is to raise μ if loss increase and reduce it otherwise, aiming to ensure convergence. However, the relationship between loss dynamics and convergence is not elaborated upon.

It is worth noting that FedAvg can be seen as a special case of FedProx when: (1) $\mu = 0$; (2) the local solver is specifically SGD; and (3) the number of local epochs E remains constant across devices and rounds.

FedNova

The *FedNova* federated algorithm, introduced by Wang et al. [51], employs a normalized averaging method that addresses objective inconsistency while ensuring fast error convergence. Designed for scenarios exhibiting client system heterogeneity, *FedNova* allows the number of local iterations τ_k to be tailored to individual devices. The local iterations for client k are defined as:

$$\tau_k = \frac{E_k n_k}{B_k} \tag{3.5}$$

where E_k is the number of epochs, n_k the number of samples used by client k in the local step, and B_k represents the batch size. This flexibility can be advantageous; for instance, devices with faster computational capabilities might be allocated more local iterations.

The variability of local updates across communication rounds might be influenced by factors like network slowdowns or stragglers. The authors wonder whether the conventional *FedAvg* aggregation (as in (3.3)) ensures convergence to a consistent stationary point in presence of heterogeneous local updates. They prove that such updates, under standard averaging, may converge not to the original objective function $F(x)$ but to an inconsistent objective $\tilde{F}(x)$. The difference between these objectives can be substantial, contingent upon τ_k values. To counteract this, *FedNova* proposes normalized and scaled local updates, based on the number of local steps, prior to global model updating. Assuming a total of K clients, with S_t being the subset of $m \leq K$ clients selected in round t, the global model weight update is given by:

$$w^{(t+1)} - w^{(t)} = \sum_{k \in S_t} \frac{p_k \tau_k^{(t)} \sum_{k \in S_t} p_k \Delta_k^{(t)}}{\tau_k^{(t)}} \tag{3.6}$$

Here, $p_k = \frac{n_k}{\sum_j n_j}$ indicates the data fraction at client k, while $\Delta_k^{(t)} = w_k^{(t,\tau_k)} - w^{(t,0)}$ stands for the local parameter alterations of client k during round t.

Dynamic sampling selector

Li et al. [28] introduced the dynamic sampling selector to address the challenges of reducing communication and computation costs in federated learning with numerous clients. Given the fixed bandwidth of the central server when gathering updates from distributed clients, bottlenecks are imminent. To mitigate this, the authors suggest a dynamic sampling method that starts with a high sampling rate and then gradually reduces it with each communication round. This approach aims to expedite convergence at the onset of federated learning by involving a greater number of clients in model aggregation. As a generalized federated model emerges, the method trims the number of clients participating in model aggregation to save on communication. Though initially resource-intensive, the required number of clients drops significantly after a few training rounds. The decaying rate of the sampling rate is chosen so that, over a given number of communication rounds, dynamic sampling incurs fewer parameter transmissions than static sampling. The proposed dynamic subsampling employs an exponential decay rate to adjust the sampling rate throughout training, expressed as:

$$R(t, \beta) = \frac{1}{\exp(\beta t)}$$

At the t-th training round, the dynamic sampling rate is given by $c = \frac{C}{\exp(\beta t)}$, where C is a predefined initial sample rate. As communication rounds progress, the sampling rate

becomes very small, sometimes resulting in fewer than one client being chosen for model aggregation. In real-world scenarios, the minimum number of selected client models is kept at two. This dynamic approach, with its varying sampling rate, distinguishes it from static sampling methods.

Power of choice

The *Power Of Choice* federated algorithm, as introduced by Jee Cho et al. [27], provides a communication- and computation-efficient client selection framework that flexibly balances between convergence speed and solution bias. While preceding works have probed into the convergence of federated learning—considering data heterogeneity, communication/computation constraints, and partial client participation—most assume unbiased client selection. In this study, the authors delve into the convergence implications of federated learning under biased client selection and elucidate the effects of such biases on convergence speed. Their findings indicate that client selection, when biased towards entities with higher local losses, accelerates error convergence.

In a cross-device FL configuration with K clients, client k possesses a local dataset B_k comprising $|B_k| = D_k$ data samples. Interconnected via a central aggregating server, the collective seeks the model parameter w which minimizes the empirical risk:

$$F(w) = \frac{1}{\sum_{k=1}^{K} D_k} \sum_{k=1}^{K} \sum_{\xi \in B_k} f(w, \xi) = \sum_{k=1}^{K} p_k F_k(w) \tag{3.7}$$

where $f(w, \xi)$ defines the composite loss function relative to sample ξ and parameter vector w. $p_k = \frac{D_k}{\sum_{k=1}^{K} D_k}$ represents the data fraction at the k-th client, while $F_k(w) = \frac{1}{|B_k|} \sum_{\xi \in B_k} f(w, \xi)$ represents the local objective function of client k.

Drawing insights from the aforementioned convergence analyses, the authors propose a *power-of-choice* client selection strategy, termed $\pi_{\text{pow}-d}$. In this strategy, the server chooses the active client set $S^{(t)}$ as follows:

1. **Sample the Candidate Client Set:** The central server samples a candidate set A of d clients ($m \leq d \leq K$) without replacement. Each client k is selected with probability p_k, the fraction of data at the k-th client.

2. **Estimate Local Losses:** The global model $w^{(t)}$ is dispatched to the clients in set A, and they compute and relay their local loss $F_k(w^{(t)})$ back to the central server.

3. **Select Highest Loss Clients:** From set A, the server constructs the active client set $S^{(t)}$ by choosing the top $m = \max(CK, 1)$ clients with the largest values of

$F_k(w^{(t)})$, with ties broken at random. These clients participate in the subsequent training round, spanning iterations $t+1, t+2, \dots t+\tau$.

The authors further propose two strategy variants:

- **Computation-efficient Variant $\pi_{\mathbf{cpow}-d}$:** To save computational resources, rather than evaluating $F_k(w)$ over the complete local dataset B_k, an estimate is derived from $\frac{1}{|\xi_{bk}|}\sum_{\xi \in \xi_{bk}} f(w,\xi)$, where ξ_{bk} is a uniformly random sampled mini-batch from B_k.
- **Communication- and Computation-efficient Variant $\pi_{\mathbf{rpow}-d}$:** Saving both computation and communication cost, chosen clients for each round send their averaged loss over local iterations (computed as $\frac{1}{\tau|\xi_k^{(l)}|}\sum_{l=t-\tau+1}^{t}\sum_{\xi \in \xi_k^{(l)}} f(w_k^{(l)},\xi)$) during the model update to the server. The most recently received value from each client, treated as a proxy for $F_k(w)$, guides client selection. For clients yet to be selected, the latest value is set to ∞.

3.1.2. Open Issues

Federated learning, as a rapidly evolving area of study, still faces many unsolved challenges, some of which are elucidated by Baresi et al. [4].

Scalability

The traditional model prescribes a centralized orchestrator that oversees the management, distribution, and aggregation of results from various clients. Such a central point can readily become a system bottleneck or a single failure point, jeopardizing the entire training process should it malfunction. The scientific community has been actively seeking decentralized federated learning solutions. Bonawitz et al. [6], for instance, propose an elastic architecture capable of scaling with the number of connected clients, although it remains conceptual since no experimental results are shown in the paper. Other research avenues like those explored by Korkmaz et al. [32] leverage peer-to-peer (P2P) networks to remove the need for an orchestrator. Here, every client undergoes local training and conveys results to selected neighboring clients, then updates its model by averaging received parameters and its local model. Still, there might be a necessity for an orchestrator to initialize the model, tasks, and training algorithms, especially during hyperparameter tuning.

Client selection

Random selection remains the go-to method used by orchestrator to select clients participating in a training round. However, more strategic selection could enhance system performance. Clients could convey nonsensitive metadata about their status, like the quality and size of local samples, computational resource availability, or battery life, to the orchestrator. With such insights, the orchestrator could more judiciously select clients, potentially mitigating the drag from stragglers (i.e., clients that are slower than the others to complete a training round) without compromising privacy. Furthermore, an orchestrator might maintain logs on device availability and historical failures to refine selection. Noteworthy initiatives in this domain include those previously discussed [27, 28].

Resource allocation

The resources available on each device can vary dramatically, for instance, between IoT devices and base stations. Therefore, expecting uniform workloads or performance across devices is unrealistic. McMahan et al. proposed FedAvg [40], performing multiple local training iterations within every global round before relaying results to the orchestrator, yielding bandwidth savings over more traditional methods like FedSGD [40]. Li et al. [46] advanced FedProx, a FedAvg variant, to exploit resource heterogeneity. Features include the local processing of variable workloads depending on available resources, the aggregations of partial straggler data, and a tunable parameter limiting the impact of local updates to prevent significant model shifts per round. Yu et al. [58] introduced Fed+ that combines various FL algorithms, letting heterogeneous devices undertake varying number of iterations during training rounds, depending on the device's resources. Baresi et al. [3] portrayed FL applications as self-adaptive systems, optimizing client resource allocation at runtime based on model accuracy and network overhead constraints. Preliminary evaluations proved the efficacy of this strategy in conserving client resources with respect to existing methods. While basic strategies like FedAvg remain agnostic to client resources, more refined methods must take into account device heterogeneity and resource availability to optimize iteration scheduling and yield superior results faster.

3.2. Analysis of FL frameworks

Setting up a realistic federated network for experimentation presents notable challenges and costs. A real federated environment often includes an extensive number of clients. For instance, private networks typically comprise hundreds to thousands of devices, while public networks can extend to millions. The device configurations within these networks

are inherently heterogeneous. Replicating such a vast and diverse environment mandates acquiring or renting a multitude of devices, resulting in substantial costs. Moreover, the geographic distribution of these devices, which leads to varying communication times, is a pivotal factor in a federated setup. Emulating this distribution physically can introduce logistical difficulties. While a physical network mirrors real conditions, its inflexibility can inhibit simulation of diverse network states. Changes in hardware capabilities, network conditions, or other uncontrollable factors mean that experimental outcomes are hardware-dependent, complicating reproducibility. The nature of data collection in a federated environment, whether through sensors or user interactions, further complicates the setup and can escalate costs. For instance, devices equipped with accelerometer sensors might be suitable for activity recognition, yet inadequate for other common federated tasks like image recognition or text prediction. Addressing this breadth requires diverse data collection methods. Given these complexities, numerous existing federated learning frameworks aim to navigate and mitigate these challenges.

3.2.1. Tensorflow federated

TensorFlow Federated (TFF) [20] is an open-source framework designed for machine learning and various computations on decentralized data. TFF is developed to promote open research and experimentation in federated learning (FL). It allows developers to simulate federated learning algorithms on their models and data, as well as to test innovative algorithms.

TFF structures its interfaces into two primary layers:

- The *Federated Core API* provides lower-level interfaces that merge TensorFlow with distributed communication operations.
- The *Federated Learning API* offers high-level interfaces, enabling integration of existing machine learning models into the TFF framework.

These layers facilitate fundamental tasks, such as federated training or evaluation, without delving deep into the intricacies of the federated learning algorithms.

To enhance experimental capabilities, TFF's creators introduced a dataset collection for simulation scenarios. These datasets enable the retrieval of non-iid partitions for a specified client count. However, a limitation lies in its rigidity; the dataset partition cannot be customized. The number of clients for partitioning the datasets is predetermined, and given a client id, its local dataset is also unchangeable. This restricts the potential to simulate environments with a varied number of clients or clients with different sample

sizes. Another drawback of this framework is that it is not very user friendly and flexible, given that most of the core logic is written in the core API, written mainly in C++, and not very well documented.

The library is open-source, with an Apache license, so friendly to expandibility, and the project hosted on Github [21] is quite popular and well maintained, counting 2.2k stars.

3.2.2. FATE

FATE [39] is an open-source project geared towards facilitating a secure and federated AI ecosystem. Its design encompasses multiple secure computation protocols, empowering big data collaboration while ensuring compliance with data protection regulations. Sponsored by WeBank, a privately-owned neo bank situated in Shenzhen, China, FATE is available for both standalone and cluster deployment configurations. The structure of FATE consists of several components, here we highlight only a few of them:

- **FATEFlow:** The central component of FATE, FATEFlow encapsulates the end-to-end machine learning orchestration pipeline. This pipeline is proficient in handling tasks ranging from data preprocessing to model training, testing, publishing, and serving.
- **FederatedML:** This component is pivotal for the implementation of numerous standard machine learning algorithms alongside other utility tools.
- **FATEBoard:** Serving as a suite of visualization and dashboarding tools, FATEBoard simplifies the process of exploring, analyzing, and comprehending federated learning models.
- **FATE Serving:** Dedicated to enabling federated learning models for production usage, this component ensures smooth deployment in real-world applications.
- **FATE-Client:** Designed to interact with various FATE components, this component plays a crucial role in system operations.

One of FATE's distinct advantages lies in the breadth of its functionality. Specifically, FATE supports live visualization through *FATEBoard*, incorporates its own model serving mechanism via *FATEServing*, is compatible with Spark clusters facilitating large-scale computations, and *KubeFATE* enables the deployment of FATE using Docker Compose and Kubernetes.

However, navigating FATE's myriad modules and options is not always intuitive. While it boasts extensive documentation, users might find it cumbersome to browse through.

Moreover, the English version occasionally appears to be incomplete.

In summary, FATE emerges as a business-ready FL framework, loaded with pre-built modules that facilitate pipeline construction and an array of additional functionalities, including orchestration. However, it is not easy to build and use custom modules and usability could be improved.

3.2.3. FedJax

FedJAX is a JAX-based open-source library designed specifically for Federated Learning simulations [45].

JAX [19], abbreviated for "Just After eXecution," is a contemporary machine learning library pioneered by DeepMind. Unlike Tensorflow, which is a flagship product of Google, JAX is more research-oriented. The research community is increasingly adopting JAX, thanks to its unique features and its similarity with the NumPy syntax. Essentially, JAX serves as a Just-In-Time (JIT) compiler that capitalizes on maximizing FLOPs to produce optimized code while retaining the elegance of pure Python. Some of the standout features of JAX include:

- Just-in-Time (JIT) compilation.
- Extension of NumPy code to CPU, GPU, and TPU.
- Capabilities for automatically obtaining of the gradient function through differentiation of a function.
- Automatic vectorization.
- Enhanced options for numerical program transformations and compositions.

The primary objective of FedJAX is to provide researchers with an easy-to-use platform that can expedite the process of developing and evaluating federated algorithms. A notable feature of this library is its compatibility with accelerators, such as GPU and TPU, without necessitating significant additional configurations. Despite its capabilities, it is important to note that FedJAX is primarily aimed at research purposes and is not suited for deployment across distributed devices.

Given that simulation is the primary concern, there is no need to introduce complexities associated with distributed machine communication. The implication is that all per-client work can efficiently run on a singular machine, resulting in a simpler API and faster execution in most scenarios.

However, despite the strong backing from a conglomerate like Google and the utilization of a robust framework like JAX, FedJAX has yet to gain significant popularity. As of now, its presence on platforms like GitHub is relatively modest, with only 242 stars [17], and it does not exhibit the active maintenance seen in other projects.

3.2.4. FedML

FedML [23] seeks to provide a comprehensive platform for ML, complemented by a suite of tools tailored to diverse needs in the domain.

At its core is the *FEDML Nexus AI*, a next-generation cloud service targeting LLMs & Generative AI. This service allows developers to perform model training, deployment, and federated learning across a number of platforms, including decentralized GPUs, multi-clouds, edge servers, and even smartphones, in a manner that's both economical and secure. Seamlessly integrated with the FEDML open-source library, FEDML Nexus AI ensures robust support across three pivotal AI infrastructure layers: user-centric MLOps, an efficiently managed scheduler, and high-performance ML libraries suitable for various AI tasks on GPU Clouds.

Among the large number of components available, the *FEDML Federate* stands out as the federated learning platform. This is further empowered by the world's first *FLOps* (Federated Learning Ops), which promotes on-device training on diverse platforms like smartphones and cross-cloud GPU servers. The platform's versatility extends to facilitating zero-code, lightweight, cross-platform, and provably secure federated learning and analytics, thanks to its cross-platform Edge AI SDK, deployable on edge GPUs, smartphones, and IoT devices.

FEDML's strength revolves around its versatile federated learning simulator, offering support for data silo-based federated learning, distributed training acceleration, MLOps, open-source backing, and an array of datasets. However, its main drawback is in its complexity; while the platform offers an extensive toolset and multiple solutions, their complexity and overlaps often lead to confusion, making the learning process challenging.

The core libraries are open source and the project counts 3200 stars on GitHub [13].

3.2.5. PySyft

PySyft [61], an initiative of the OpenMined project backed by major tech corporations and open-source contributors, is a promising open-source federated learning (FL) framework. From a larger perspective, *PySyft* is not only a FL framework but a remote data

science platform, facilitating experiments on sensitive data. This is achieved while ensuring privacy via mechanisms like differential privacy and secure multi-party computation.

For optimal functioning, *PySyft* integrates with *PyGrid*, a bridge between data owners (FL clients) and data scientists (FL servers). While *PyGrid* is fundamentally a Linux application, it offers cross-OS compatibility through containers, and its synergy with *HAGrid*, a CLI tool, augments deployment ease.

When performing FL experiments *PySyft* is not framework-agnostic, but supports only the deep learning libraries PyTorch and TensorFlow. *PySyft* currently faces challenges in maintaining documentation congruency with its evolving versions, potentially due to its active development by its open source community.

In summary, *PySyft* goes beyond normal FL frameworks by creating a platform where not only FL but also federated data science is addressed. Although it offers multiple capabilities, which can occasionally lead to complexity, especially in its FL segment, the framework's current documentation gaps makes it difficult to understand by newcomers. Yet, its potential is undeniably evident, as echoed by its impressive traction, with over 9000 stars on GitHub [41].

3.2.6. Flower

Flower (flwr) [5] is a versatile framework designed for building federated learning systems, rooted in principles that prioritize customization, extensibility, framework-agnosticism, and maintainability.

Developed from a research project at the University of Oxford, Flower accommodates diverse federated learning configurations, enabling users to tailor the system to their specific use cases. Its framework-agnostic nature allows seamless integration with various machine learning libraries, supporting PyTorch, TensorFlow, and more.

Flower's server orchestrates the federated learning process, guided by a user-defined strategy, while clients, represented by subclasses of `flwr.client.Client`, await server instructions for training and evaluation.

Notably, Flower offers a powerful simulation capability through the *VirtualClientEngine*, facilitating experimentation and validation across different scenarios. With a well-documented API, an array of tutorials, and an active community, Flower stands out as a comprehensive and accessible tool for researchers and developers in the federated learning domain. Its baselines, mirroring state-of-the-art techniques, further enhance its utility for comparative studies and experimentation.

3.3. Framework of choice

In this section, we clarify the selection of our federated learning framework, outlining the reasons behind our choice. We then present the chosen framework in detail.

3.3.1. Criteria

In this subsection, we elaborate on the criteria that have guided our choice of selecting *Flower* as the framework to use in this thesis.

In Table 3.1 we give a consolidated view of the strengths and weaknesses of the aforementioned federated learning frameworks.

User friendliness

Flower is easy to use and learn. The primary objective is to utilize a framework that other researchers can easily reuse to reproduce our results, and *Flower* fits this requirement. Its well-maintained documentation, tutorials, and baselines provide a structure for implementing our techniques and ensuring their reusability. Compared to other powerful frameworks such as *FedML* and *PySyft*, *Flower* excels in its documentation, aligning perfectly with our needs.

Extendability

Flower offers numerous classes that can be easily extended and customized with minimal effort. This flexibility allows us to concentrate on the core algorithms and techniques instead of integrating changes into a convoluted structure. This criterion led us to discard *Tensorflow Federated*, as it heavily relies on the underlying *Federated Core API*. This API is not thoroughly documented, making it challenging to understand and resulting in intricate, hard-to-extend code.

Capabilities

Flower is among the most versatile frameworks available. Its capabilities range from adaptability to various client types (mobile, embedded, and edge devices) to the monitoring of the FL loop using logging and external tools like *Grafana*. Its compatibility with all major machine learning frameworks ensures its suitability for a range of tasks, including image classification, natural language processing, and sentiment analysis.

Framework	Strengths	Weaknesses
TensorFlow Federated (TFF)	+ Integration with TensorFlow. + Low-level and high-level APIs. + Simulation scenarios with predefined datasets.	- Limited dataset customization in simulation. - Core logic in C++, less user-friendly.
FATE	+ Secure computation protocols. + End-to-end ML orchestration pipeline. + Versatility in deployment options.	- Framework complexity. - Documentation complexity. - English version occasional incompleteness.
FedJax	+ JAX-based for easy NumPy syntax. + Compatible with GPU and TPU.	- Limited popularity and maintenance. - Primarily research-oriented. - Not suited for deployment across distributed devices.
FedML	+ Support for various AI infrastructure layers. + Business-ready with pre-built modules.	- Complexity in module usage. - Overlapping functionalities.
PySyft	+ Offers multiple capabilities. + Remote data science platform.	- Documentation gaps - Can occasionally lead to complexity. - Limited framework support.
Flower	+ Customizable and extendable. + Framework-agnostic and versatile. + Well-documented with active community. + Built-in simulation engine.	- None significant.

Table 3.1: Strengths and weaknesses of FL frameworks.

Simulation

Flower's built-in simulation engine allows for quick prototyping and verification of each strategy's results without adapting or scaling the simulation code. Varying hyperparam-

eters and running simulations with a large number of clients, even on a single machine, becomes feasible. This adaptability enables experimentation with different configurations without limitations imposed by the tool.

Popularity

Lastly, the chosen framework should be popular and actively maintained. This characteristic is vital to ensure that the framework remains updated and functional so that researchers can reproduce implemented results even years later. Thanks to the variety of channels *Flower* offers for community interaction, it has gained considerable popularity. Most importantly, its community is highly active and engaged.

In summary, the choice of *Flower* is motivated by its user-friendliness, extendability, broad capabilities, built-in simulation engine, and active community support. It excels in providing a comprehensive and flexible platform for federated learning, making it suitable for our research and experimentation needs.

3.3.2. Flower in detail

In this subsection, we delve into Flower, highlighting its numerous features and capabilities.

Flower (flwr) is a framework for building federated learning systems [5]. The design of Flower is based on a few guiding principles:

1. **Customizable:** Federated learning systems vary wildly from one use case to another. Flower allows for a wide range of different configurations depending on the needs of each individual use case.

2. **Extendable:** Flower originated from a research project at the University of Oxford, so it was built with AI research in mind. Many components can be extended and overridden to build new state-of-the-art systems.

3. **Framework-agnostic:** Different machine learning frameworks have different strengths. Flower can be used with any machine learning framework, for example, PyTorch, TensorFlow, Hugging Face Transformers, PyTorch Lightning, MXNet, scikit-learn, JAX, TFLite, fastai, Pandas for federated analytics, or even raw NumPy for users who enjoy computing gradients by hand.

4. **Understandable:** Flower is written with maintainability in mind. The community is encouraged to both read and contribute to the codebase.

Being complete customization one of the founding principles behind Flower, the framework allows programmers to load the data, define the model and configure its training and evaluation phase at each client independently, offering a supporting set of classes to integrate all of this into a federated learning environment. Federated learning systems consist of a server and multiple clients, each of them do have a counterpart class in Flower.

Server

In federated learning, the server sends the global model parameters to the clients, which train their models on the local datasets and send the updated model parameters back to the server. As a result the role of the server is to orchestrate this process along all its phases.

It is possible to implement a federated learning server in Flower by implementing subclasses of `flwr.server.Server`. This class uses some other objects to delegate part of its functionality, in detail:

- `ClientManager`: Manages a set of `ClientProxy` objects, each representing a single client connected to the server, and selects clients participating in each training round depending on the logic defined in the *Strategy*.
- `Strategy`: Represents a FL algorithm and defines the logic behind each phase: selection of clients, configuration of federated training, aggregation of weights returned to the server into the global model.

In Figure 3.2, the main components of the FL server in Flower are presented through a UML class diagram.

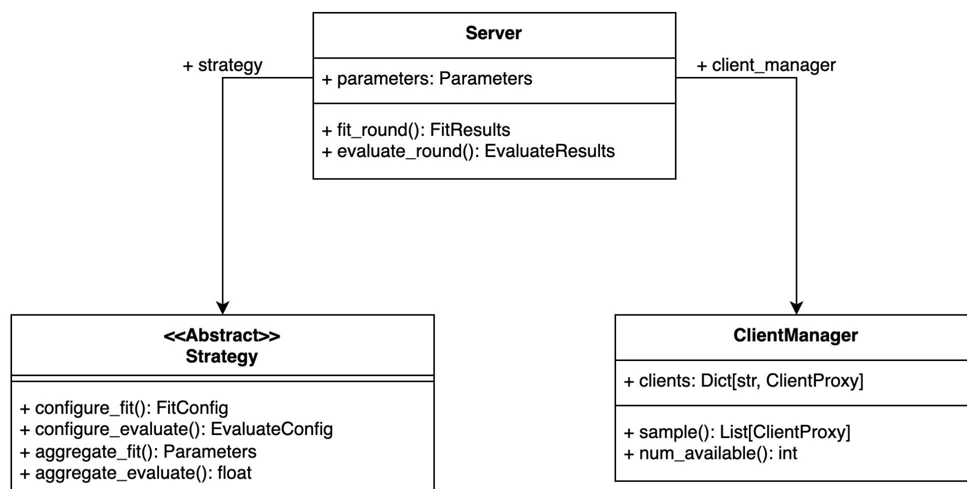

Figure 3.2: Flower server class diagram.

In summary, the FL loop in the *Server* asks the *Strategy* to configure the next round of FL, sends those configurations to the affected clients, receives the resulting client updates (or failures) from the clients, and delegates result aggregation to the *Strategy*. It takes the same approach for both federated training and federated evaluation, with the added capability of server-side evaluation (again, via the *Strategy*).

Clients

The client side is simpler in the sense that it only waits for messages from the server. It then reacts to the messages received by calling user-provided training and evaluation functions.

In Flower, it is possible to create clients by implementing subclasses of `flwr.client.Client`. Implementing a client, in its simplest case, involves implementing the three main methods that define the actions of a client, `get_parameters`, `fit`, and `evaluate`:

- `get_parameters`: Return the current local model parameters.
- `fit`: Receive model parameters from the server, train the model parameters on the local data, and return the (updated) model parameters to the server.
- `evaluate`: Receive model parameters from the server, evaluate the model parameters on the local data, and return the evaluation result to the server.

Federated learning systems have multiple clients, so each client will be represented by its own instance of *FlowerClient*. If we have, for example, three clients in our workload, then we would have three instances of *FlowerClient*. Flower calls `FlowerClient.fit` on the respective instance when the server selects a particular client for training (and `FlowerClient.evaluate` for evaluation).

To train the model, each client needs its own dataset. Depending on the application, Flower allows clients to load datasets from popular datasets that have been previously appropriately preprocessed and split in partitions, or to load their own dataset stored locally.

As previously mentioned, Flower is designed to be framework-agnostic. This means it supports various machine learning frameworks. Not only can users build models used by clients with popular deep learning libraries like PyTorch and TensorFlow, but they can also utilize frameworks such as TensorFlow Lite for mobile clients [15].

Simulation

As already elaborated in Section 3.2, simulating Federated Learning workloads is useful for a multitude of use-cases: we might want to run our workload on a large cohort of clients but without having to source, configure and manage a large number of physical devices; we might want to run our FL workloads as fast as possible on the compute systems we have access to without having to go through a complex setup process; we might want to validate our algorithm on different scenarios at varying levels of data and system heterogeneity, client availability, privacy budgets, etc.

Flower can accommodate these scenarios by means of its *VirtualClientEngine*. The *VirtualClientEngine* schedules, launches and manages virtual clients. These clients are identical to non-virtual clients in the sense that they can be configured by creating a class inheriting, for example, from `flwr.client.NumPyClient` and therefore behave in an identical way. In addition to that, clients managed by the VirtualClientEngine are:

- *resource-aware*: this means that each client gets assigned a portion of the compute and memory on the system. The user can control this at the beginning of the simulation and it allows to control the degree of parallelism of the Flower FL simulation. The fewer the resources per client, the more clients can run concurrently on the same hardware.
- *self-managed*: this means that the user do not need to launch clients manually, instead this gets delegated to VirtualClientEngine's internals.
- *ephemeral*: this means that a client is only materialized when it is required in the FL process (e.g. to do `fit()`). The object is destroyed afterwards, releasing the resources it was assigned and allowing in this way other clients to participate.

The *VirtualClientEngine* implements virtual clients using *Ray*, an open-source framework for scalable Python workloads. In particular, Flower's *VirtualClientEngine* makes use of actors to spawn virtual clients and run their workload.

In Figure 3.3 the Flower core framework architecture with both Edge Client Engine and Virtual Client Engine is shown. With edge clients we refer to clients living on real edge devices and communicating with the server over RPC. Virtual clients on the other hand are those managed by the VCE for simulation on a single machine, they consume close to zero resources when inactive and only load model and data into memory when the client is being selected for training or evaluation.

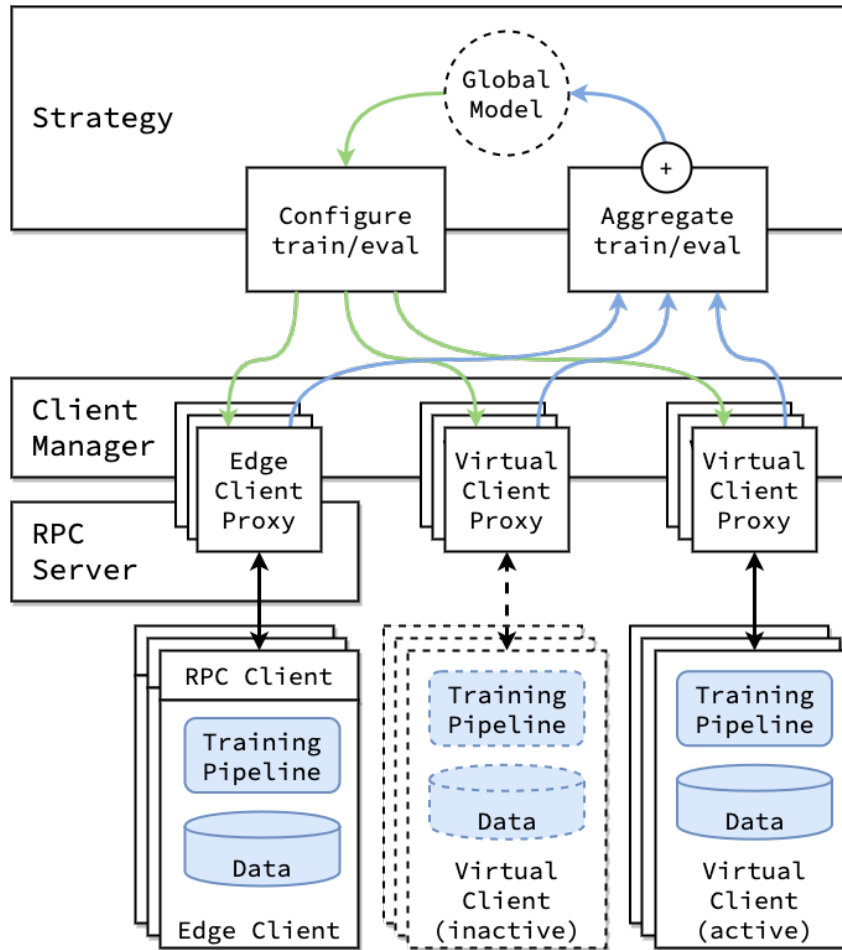

Figure 3.3: Flower core framework architecture.

Baselines

Among its many advantages, Flower provides a set of baselines, which are organized directories designed to reproduce results from notable publications or benchmarks. These baselines enable researchers and students to effortlessly replicate state-of-the-art FL techniques and algorithms, facilitating comparisons with their own ideas and experiments. Flower actively encourages its open-source community to expand the available baselines, providing scripts and abstract classes to integrate new techniques.

The assortment of FL baselines in Flower is continuously expanding, currently featuring more than five cutting-edge techniques ready for experimentation. These include DASHA [49], FedMeta [10], FedMLB [31], FedProx [46], Fjord [24], MOON [36], and TAMUNA [11].

This feature of Flower echoes what was highlighted in Section 3.2 regarding the primary benefits of a FL framework: the ability to compare new methodologies with state-of-the-

art techniques.

Documentation

Another significant advantage of Flower is its up-to-date and well-organized documentation. Both APIs and classes are meticulously documented and maintained. Such clarity is not a given, especially for an open-source project boasting over 90 contributors. This is achieved by enforcing a rigorous coding and documentation style, which is verified using analyzers that must be satisfied before merging new code into the repository.

Alongside its great documentation, Flower provides an array of well-crafted tutorials to assist newcomers in familiarizing themselves with the principles of Federated Learning as well as the components and classes of Flower. These tutorials include ready-to-run Colab notebooks. Users begin by constructing their initial system in Flower and progressively evolve it into a sophisticated Federated Learning environment.

Lastly, Flower sustains an active community through various channels: a dynamic and organized Slack channel where both novices and experts can pose questions, suggest new features, and directly interact with the project's founders. Additionally, they host events such as Flower Monthly [14], where community members can showcase new applications of Flower across diverse sectors, spanning from finance to medicine.

4 | Implementation

This chapter describes our proposed solutions to address the challenges introduced in Section 1.1. The chapter is organized as follows: in Section 4.1, the solutions implemented for the topic of dynamic selection of clients are presented; in Section 4.2, the solutions implemented for the topic of resource-aware workload allocation are presented.

4.1. Dynamic selection

The focus of this section is on algorithms and techniques designed for the rational dynamic selection of clients that participate in each round of training and/or evaluation in federated learning.

This emphasis addresses one of the open issues in federated learning as highlighted in Section 3.1.2. While random selection remains the prevalent strategy employed by the orchestrator to determine the clients participating in a training round, a more strategic selection could further optimize system performance.

To enhance the selection process, techniques that involve clients providing non-sensitive metadata regarding their current status need to be developed. This metadata should encompass factors such as the quality and size of local samples, computational resource availability, or even battery life. With these insights at its disposal, the orchestrator can make more informed choices about client selection, thus speeding up the convergence of training or maintaining similar performances but using fewer clients at each round, thereby saving resources. Moreover, by maintaining logs on device availability and tracking historical failures, the orchestrator can further fine-tune its selection strategy.

The contributions of this thesis on this topic, as underscored in Section 1.2, are threefold:

1. Investigation of techniques for dynamic selection of clients.
2. Extension of the Flower framework with dynamic sampling strategies for ease of reproducibility by other researchers and consolidated simulation.
3. Integration with resource-aware workload allocation strategies.

In total, five state-of-the-art strategies were implemented and integrated into the Flower framework to facilitate comparison and to ensure straightforward reproducibility: *FedAvg*, *dynamic sampling*, *pow-d*, *cpow-d* and *rpow-d*. Figure 4.1 depicts a UML class diagram of these techniques.

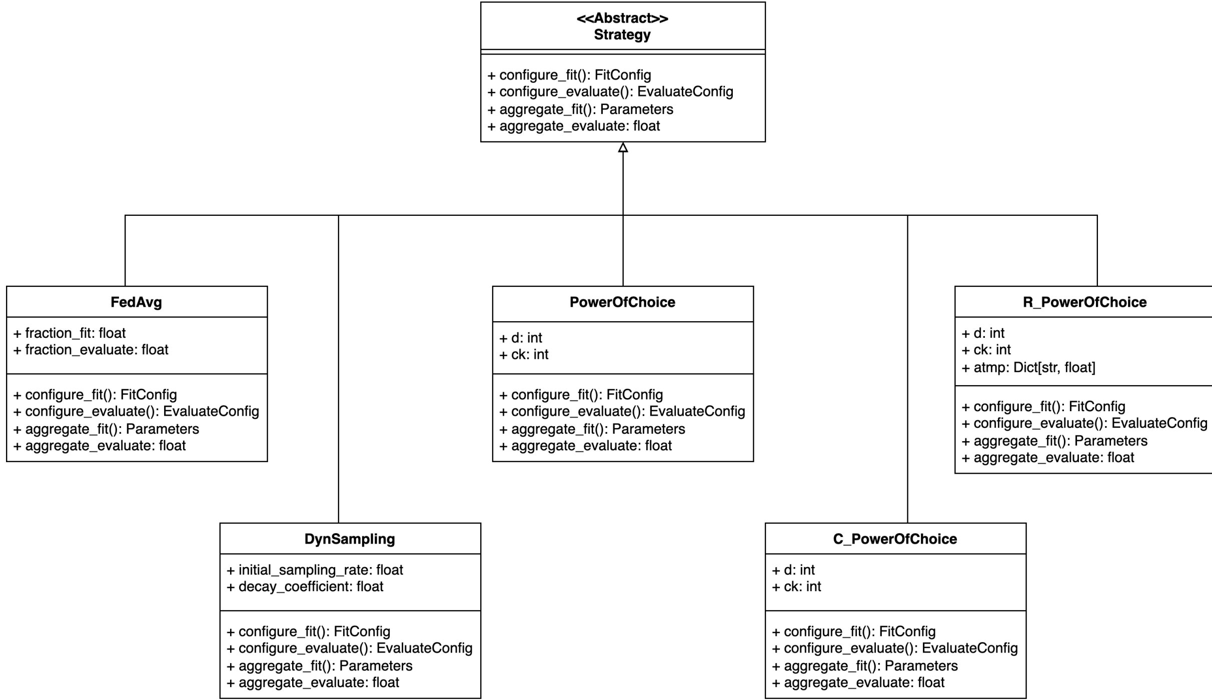

Figure 4.1: The five strategies implemented.

4.1.1. FedAvg

FedAvg, introduced by McMahan et al. [40] and detailed in Section 3.1.1, is often used as representation of a random selector. It is universally recognized as the benchmark for client selection in federated learning.

Consider a setting with a predetermined set of K clients. Each client k has a unique local dataset of size n_k. At the beginning of every round, this strategy consists of selecting a random subset S_t of clients, which represents a fraction C of the total clients. In Table 4.1, the parameters of this strategy are specified.

Parameter	Description
T	Total number of rounds
K	Total number of clients
C	Fraction of clients to select at each round

Table 4.1: Parameters of the FedAvg technique

The pseudocode for the FedAvg strategy is presented in Algorithm 4.1.

Algorithm 4.1: FedAvg Strategy

```
parameters = initialize_parameters()
for round in t = 1, 2, ..., T :
    # Sample m clients at random
    m = max(C * K, min_num_clients)
    clients = random.sample(m)

    # Use the sampled clients for training
    for client in clients:
        fit_result = client.fit()
        fit_results.append(fit_result)

    # Aggregate fit results using a weighted average
    parameters, results = aggregate_fit(fit_results)

    # Perform server-side evaluation
    evaluate(parameters, test_set)
```

Parameters and results are aggregated using the formula:

$$w_{t+1} = \frac{\sum_{k=1}^{K} n_k \cdot w_{k,t+1}}{n}$$

where the weights are determined by the dataset size n_k of each client.

Each client executes a round of training on its local dataset when the `fit()` function is called.

Flower provides a built-in *FedAvg* implementation among its baselines, ensuring seamless integration for new projects wishing to benchmark against *FedAvg*. In particular, the core strategy was already implemented, and we integrated it with our specific client, using custom models and datasets. *FedAvg* serves as our baseline for comparing new techniques.

4.1.2. Dynamic sampling

One of the primary constraints of *FedAvg* is its method of selecting clients for training. In each training round, it selects a number m of clients, where $m = \max(C \cdot K, 1)$. Here, C and K respectively represent the fraction of clients to select and the total number of clients. Both of these variables are predefined and remain static throughout the federated learning process.

Our focus in this thesis is to explore dynamic techniques that modify client selection dynamically at each round, rather than sticking to the static nature of *FedAvg*.

For this reason we deepen the *Dynamic Sampling* technique. This method, first introduced by Ji et al. [28] and detailed in Section 3.1.1, was developed to address the communication and computation challenges in federated learning, especially with a large number of clients.

The core idea behind this technique is a dynamic sampling method. Initially, it starts with a high sampling rate, which reduces gradually with each communication round. This strategy seeks to accelerate convergence at the beginning by involving more clients in model aggregation. As a more generalized federated model begins to take shape, the method reduces the number of clients in the aggregation process, thus conserving communication resources. While it might be resource-intensive at first, the number of participating clients drops significantly after a few training rounds. The decay rate of the sampling rate is chosen to ensure that, over multiple communication rounds, dynamic sampling involves fewer parameter transmissions than its static counterpart.

The dynamic subsampling utilizes an exponential decay rate for adjusting the sampling rate during training. This can be represented as:

$$R(t, \beta) = \frac{1}{\exp(\beta t)}$$

For the t-th training round, the dynamic sampling rate is defined by:

$$c = \frac{C}{\exp(\beta t)}$$

Where C is a pre-defined initial sample rate. As communication rounds progress, this sampling rate becomes very small, sometimes even selecting fewer than one client for model aggregation. However, in practical applications, a minimum of two client models are chosen for aggregation. This dynamic approach, with its variable sampling rate, differentiates this strategy from static sampling techniques.

Although the core idea of the strategy has been implemented exactly as in the aforementioned paper, extending Flower to support this logic involved creating a set of custom classes. In particular, we created a custom `Strategy` that implements the core algorithm of selection, integrated it into a custom `Server`, and with a custom `Client`, using our own models and datasets. Finally, we allowed integration of this strategy with resource-aware workload allocation techniques, which will be presented shortly.

A number of parameters are also involved in this technique, and they are reported in Table 4.2. All of them have been defined in the `conf.yaml` file and can be easily changed for new experiments.

Parameter	Description
T	Total number of rounds
K	Total number of clients
C	Initial sampling rate
β	Decay rate for adjusting the sampling rate during training

Table 4.2: Parameters of the Dynamic Sampling technique

The core of the strategy involves dynamically modifying the number of clients sampled at each round accordingly to an exponential decay rate starting from the initial sampling rate.

The pseudocode for this part of the strategy is depicted in Algorithm 4.2.

Algorithm 4.2: Dynamic sampling Strategy

```
parameters = initialize_parameters()
for server_round in t = 1,2,...,T :
    # Compute sample rate c
    sample_rate = C / math.exp(β * server_round)

    # Compute number of clients to sample
    sample_size = max(sample_rate * K, min_fit_clients)

    # Sample sample_size < K clients at random
    clients = random.sample(sample_size)

    # Use the sampled clients for training
    for client in clients:
```

```
        fit_result = client.fit()
        fit_results.append(fit_result)

    # Aggregate fit results using a weighted average
    parameters, results = aggregate_fit(fit_results)

    # Perform server-side evaluation
    evaluate(parameters, test_set)
```

The remaining part of the technique involves pretty standard clients that load their own dataset partition and train a model, in our case implemented using Tensorflow.

4.1.3. Power of choice

Dynamic sampling introduces dynamic selection of clients, however it still uses a random criteria to select clients at each round. As highlighted at the beginning of this Section, we would like to experiment some techniques that select clients dynamically depending on how do they perform during training.

Here comes *Power of Choice*, a technique for selecting clients introduced by Jee Cho et al. [27], that strives to provide a communication- and computation-efficient client selection framework that flexibly balances between convergence speed and solution bias. Power of Choice consists of a two key characteristics:

- Dynamically selects clients that have the highest loss during training, at the cost of obtaining a biased client selection technique.
- Aims for a faster convergence speed by selecting clients with the highest loss, thus enabling the model to converge more quickly.

The idea behind this selection technique is that clients having the highest loss will be those who still have something "to contribute" to the global model, as a result it is convenient selecting them to make the global model faster converge at the cost of having a biased model.

Three variants of this strategy are proposed in the corresponding paper: *pow-d*, *cpow-d* and *rpow-d*. We extended Flower to include of all of them. Being Power of Choice a more complex strategy, it required to create a custom Server for each technique, as well

as a custom Strategy and other custom classes such as Criterion to customize the logic to select the clients as well as custom clients to behave differently depending on the variant employed.

In Table 4.3, the parameters of the three variants are introduced.

Parameter	Variant	Description
T	*all*	Total number of rounds
K	*all*	Total number of clients
C	*all*	Fraction of clients to select at each round
d	*all*	Number of clients to select in the candidate set
b	*cpow-d*	Number of samples in the mini-batches used in *cpow-d*

Table 4.3: Parameters of the Power of Choice techniques

pow-d

This strategy is the base variant of the *Power of Choice* techniques. It consists of the following phases for server-side selection of active clients at the start of each round:

1. **Sample the Candidate Client Set:** The central server samples a candidate set A of d clients ($m \leq d \leq K$) without replacement. Each client k is selected with probability p_k, the fraction of data at the k-th client.
2. **Estimate Local Losses:** The global model $w^{(t)}$ is sent to the clients in set A, and they compute and return their local loss $F_k(w^{(t)})$ to the server. This phase is denoted as `evaluate_first_phase` in the pseudocode.
3. **Select Highest Loss Clients:** From set A, the server constructs the active client set $S^{(t)}$ by choosing the top $m = \max(CK, 1)$ clients with the largest values of $F_k(w^{(t)})$, with ties broken at random. These clients participate in the subsequent training round.

The pseudocode for this variant is presented in Algorithm 4.3.

Algorithm 4.3: pow-d Strategy

```
parameters = initialize_parameters()
for server_round in t = 1,2,...,T :

    # Select candidate set of d clients, each client has
    # probability of being chosen proportional to its
```

```
    # dataset size
    cl_prob = [num_data_samples[cid] for cid in clients]
    cl_prob_norm = [p/sum(cl_prob) for p in cl_prob]
    candidate_set = np.random.choice(clients, size=d,
        replace=False, p=cl_prob_norm)

    for client in candidate_set
        # Estimate local losses of the candidate set
        res_first_phase = evaluate_first_phase()

    # Sort clients based on their losses
    sorted_client_losses = sorted(res_first_phase)

    # Take the m clients with the highest losses
    m = max(C * K, 1)
    chosen_clients = [client in sorted_client_losses[:m]]

    # Use the sampled clients for training
    for client in chosen_clients:
        fit_result = client.fit()
        fit_results.append(fit_result)

    # Aggregate fit results using a weighted average
    parameters, results = aggregate_fit(fit_results)

    # Perform server-side evaluation
    evaluate(parameters, test_set)
```

cpow-d

The *Power of Choice* base strategy (*pow-d*) presents two primary disadvantages:

- It necessitates a preliminary phase where each client evaluates the entire local dataset, leading to increased computational overhead.
- All clients must communicate their local losses to the server in every round, intro-

ducing an additional communication cycle into the federated learning loop, resulting in increased communication costs.

To mitigate the first issue, a computationally efficient variant, from now on referred to as *cpow-d*, is proposed. The primary distinction between *pow-d* and *cpow-d* is that, instead of requiring clients to evaluate their local loss F_k on the full local dataset B_k, *cpow-d* has clients compute F_k on a mini-batch of b samples, randomly selected from B_k. This approach saves computational resources and accelerates the overall process, as the initial phase completes more quickly. However, this efficiency comes at the cost of potentially reducing the representativeness of the loss, which might not reflect the loss across the entire dataset.

The pseudocode for the *cpow-d* variant is outlined in Algorithm 4.4.

Algorithm 4.4: cpow-d Strategy

```
parameters = initialize_parameters()
for server_round in t = 1,2,...,T :

    # Select candidate set of d clients, each client has
    # probability of being chosen proportional to its
    # dataset size
    cl_prob = [num_data_samples[cid] for cid in clients]
    cl_prob_norm = [p/sum(cl_prob) for p in cl_prob]
    candidate_set = np.random.choice(clients, size=d,
        replace=False, p=cl_prob_norm)

    for client in candidate_set
        # Estimate local losses of the candidate set
        # on a mini batch
        res_first_phase = eval_first_phase_mini_batch()

    # Sort clients based on their losses
    sorted_client_losses = sorted(res_first_phase)

    # Take the m clients with the highest losses
    m = max(C * K, 1)
    chosen_clients = [client in sorted_client_losses[:m]]
```

```
    # Use the sampled clients for training
    for client in chosen_clients:
        fit_result = client.fit()
        fit_results.append(fit_result)

    # Aggregate fit results using a weighted average
    parameters, results = aggregate_fit(fit_results)

    # Perform server-side evaluation
    evaluate(parameters, test_set)
```

rpow-d

The other variant of *Power of Choice* strategy, proposed to address both the identified weaknesses of the base strategy, is denoted as *rpow-d*.

This strategy eliminates the initial phase by having selected clients return their cumulative averaged loss from local iterations when they transmit their local models to the server. The server utilizes the most recently received loss value from each client as a proxy for the loss to be used in client selection. For clients that have not yet been selected, the most recent loss value is set to ∞.

The pseudocode for the *rpow-d* variant is detailed in Algorithm 4.5.

Algorithm 4.5: rpow-d Strategy

```
parameters = initialize_parameters()
for server_round in t = 1, 2, ..., T :

    # Sort clients based on their latest losses
    sorted_client_losses = sorted(latest_losses)

    # Take the m clients with the highest losses
    m = max(C * K, 1)
```

```
    chosen_clients = [client in sorted_client_losses[:m]]

    # Use the sampled clients for training
    for client in chosen_clients:
        fit_result = client.fit()
        fit_results.append(fit_result)

        # Evaluate loss on each selected client
        eval_result = client.evaluate()
        eval_results.append(eval_result)

    # Aggregate fit results using a weighted average
    parameters, results = aggregate_fit(fit_results)


    # Update latest losses
    latest_losses.update(eval_results)

    # Perform server-side evaluation
    evaluate(parameters, test_set)
```

4.2. Resource-aware workload selection

The focus of this section is on algorithms and techniques for federated learning workload assignment that is aware of resources of clients and dynamically adapts to their variations.

As introduced in Section 3.1.2, one of the open issues of federated learning is that the resources available on each device can vary dramatically, for instance, between IoT devices and base stations. Therefore, expecting uniform workloads or performance across devices is unrealistic. Existing strategies hardly consider resource heterogeneity within their vision. Notable examples in this direction are Li et al. [46], who advanced FedProx, a FedAvg variant, to exploit resource heterogeneity. Features include the local processing of variable workloads depending on available resources, the aggregations of partial straggler data, and a tunable parameter limiting the impact of local updates to prevent

significant model shifts per round. Yu et al. [58] introduced Fed+ that combines various FL algorithms, letting heterogeneous devices undertake varying number of iterations during training rounds, depending on the device's resources. Baresi et al. [3] portrayed FL applications as self-adaptive systems, optimizing client resource allocation at runtime based on model accuracy and network overhead constraints.

While basic strategies like FedAvg remain agnostic to client resources, more refined methods must take into account device heterogeneity and resource availability to optimize workload scheduling and yield superior results faster.

In this thesis, we introduce four innovative so-called *Global Update Optimizers*, that can be thought of as extensions of strategies to tune workload allocation at each client at the beginning of each round.

To extend `Flower` to include such logic, we exploited the `configure_fit` parameter that can be passed to each `Strategy`, that is a function called at the beginning of each round by the Flower server, so that it sends a custom `FitIns` object (fit instructions) to customize the workload to be performed by each client participating in the training round. Then each *Global Update Optimizer* is a class that has a method `get_configure_fit_fn()` to get the function object to be passed to the `Strategy`.

To customize the workload assigned to each client, we varied three parameters that influence the amount of computation required at each round for each client.

- *number of epochs*: the number of complete passes through the training dataset to be performed by a client during a round.
- *batch size*: the number of training samples to work through before the model's parameters are updated.
- *fraction of samples*: the fraction of the training dataset used by a client during a round.

Depending on these three parameters, the workload for each client can be formulated as:

$$\frac{\text{training set size} \times \text{fraction of samples} \times \text{epochs}}{\text{batch size}}$$

Therefore, by varying these parameters, we vary the amount of workload assigned to each client during a round.

In total, four Global Update Optimizers were implemented and integrated into the Flower framework to facilitate comparison and to ensure straightforward reproducibility:

static optimizer, *uniform* optimizer, *Round Time* optimizer, *Equal Computation Time* optimizer. Figure 4.2 depicts a UML class diagram of these techniques.

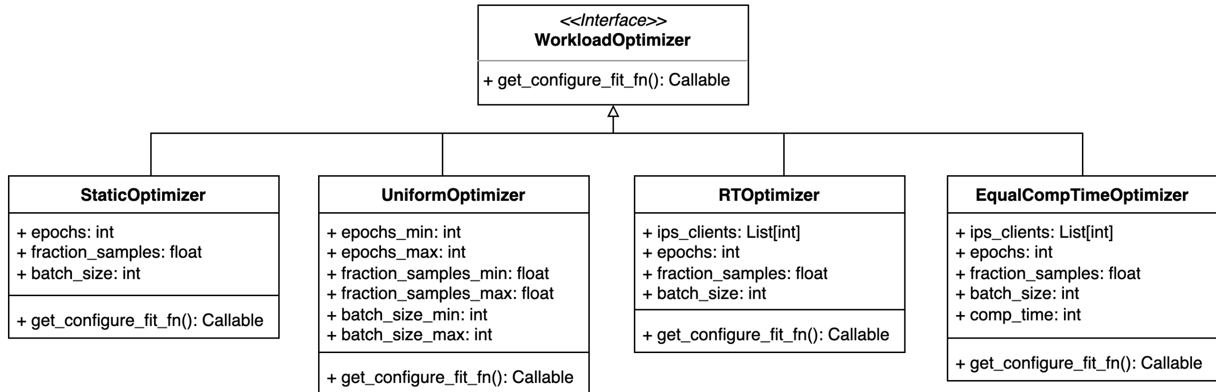

Figure 4.2: The four workload optimizers implemented

4.2.1. Static optimizer

The *Static Optimizer* statically sets *epochs*, *batch_size* and *fraction_samples* from the parameters *epochs*, *batch_size*, and *fraction_samples* in the configuration file. The parameters assigned are equal for each device k. This setting is straightforward to implement but does not take into account the specific features of devices. It is considered here as the baseline to compare with more sophisticated techniques, as it does not consider resource heterogeneity and lacks any dynamic behavior. The configuration file is a YAML file, loaded using Hydra [26]. The pseudocode of the corresponding `configure_fit` function is shown in Algorithm 4.6.

Algorithm 4.6: Static optimizer

```python
def configure_fit(server_round: int):
    # Load parameters from configuration file cfg
    # and return them
    config["epochs"] = cfg.epochs
    config["batch_size"] = cfg.batch_size
    config["fraction_samples"] = cfg.fraction_samples

    return config
```

4.2.2. Uniform optimizer

The *Uniform Optimizer* generates *epochs*, *batch_size*, and $fraction_samples$ using a uniform distribution within a specified range. The extremes of the range from which the numbers are drawn are set in the YAML configuration file.

The number of epochs for device k is extracted uniformly in the range

$$[epochs_min, epochs_max)$$

where $epochs_min$ and $epochs_max$ represent the minimum and the maximum number of epochs, respectively:

$$epochs_k \sim \mathcal{U}(epochs_min, epochs_max) \tag{4.1}$$

The batch size for device k is extracted uniformly in the range

$$[batch_size_min, batch_size_max)$$

where $batch_size_min$ and $batch_size_max$ represent the minimum and the maximum batch size, respectively:

$$batch_size_k \sim \mathcal{U}(batch_size_min, batch_size_max) \tag{4.2}$$

The fraction of samples for device k is extracted uniformly in the range

$$[fraction_samples_min, fraction_samples_max)$$

where $fraction_samples_min$ and $fraction_samples_max$ are the parameters representing the minimum and the maximum fraction of samples to use, respectively:

$$fraction_samples_k \sim \mathcal{U}(frac_samples_min, frac_samples_max) \tag{4.3}$$

This optimizer is designed to simulate the heterogeneity of the computational workload assigned to devices.

The pseudocode of the corresponding `configure_fit` function is shown in Algorithm 4.7.

Algorithm 4.7: Uniform optimizer

```
def configure_fit(server_round: int):
    # Sample the parameters for each client
    for client in clients:

        config[client]["epochs"] =
            np.random.uniform(epochs_min, epochs_max)
        config[client]["batch_size"] = np.random
            .uniform(batch_size_min, batch_size_max)
        config[client]["fraction_samples"] = np.random
            .uniform(frac_samples_min, frac_samples_max)

    return config
```

4.2.3. RT optimizer

The *RT (Round Time) Optimizer* sets parameters $batch_size$ and $fraction_samples$ from the YAML configuration file.

The number of epochs for client k is instead assigned proportionally to the computational power of client k. To model the computational power of clients, we used a parameter called IPS, acronym for *iterations per second*. The device with the highest IPS (which has the lowest computation time) is assigned with the highest number of epochs, corresponding to the value of *epochs* in the configuration file.

This optimization technique leverages the fastest devices more effectively by allocating additional computational burden to them. Consequently, devices with slower computation times no longer cause slowdowns, as their assigned number of epochs is proportionally reduced.
This technique represents an example of resource-aware workload allocation, with the additional dynamic capability to adjust workload allocation in response to variations in clients' IPS.

The pseudocode of the corresponding `configure_fit` function is shown in Algorithm 4.8.

Algorithm 4.8: RT optimizer

```
def configure_fit(server_round: int):
    ips_clients = []

    # Retrieve IPS of sampled clients
    for client in clients:
        properties_res = client.get_properties()
        ips = properties_res.properties["ips"]
        ips_clients.append((client, ips))

    # Compute the maximum IPS
    max_ips = max(ips_clients, key=lambda x: x[1])[1]

    for client, ips in ips_clients:
        # Compute scaling factor
        scale_factor = ips / max_ips

        if(ips == max_ips):
            # Set epochs to value from configuration
            epochs = cfg.epochs
        else:
            # Set epochs proportionally to scale factor
            epochs = max(1, cfg.epochs * scale_factor)

        config[client] = {
            "epochs": epochs,
            "batch_size": cfg.batch_size,
            "fraction_samples": cfg.fraction_samples
        }
    return config
```

4.2.4. Equal computation time optimizer

The *Equal Computation Time (ECT) Optimizer* generates *epochs*, *batch_size*, and *fraction_samples* depending on a fixed amount of desired computation time and the

specific IPS value for each client k.

The parameter *comp_ time* represents the computation time in seconds that each device should spend to perform the local computations at each round.

To match the computation time on each device, the server assigns a number of local iterations proportional to the IPS each client. For a client k with IPS value IPS_k, the number of iterations to perform is computed as:

$$local_iterations_k = comp_time \cdot IPS_k \tag{4.4}$$

As seen in Section 4.2, the number of local iterations is influenced by all the three parameters $epochs$, $batch_size$, and $fraction_samples$. Given the number of local iterations $\text{local_iterations}_k$, the ECT optimizer allows to vary one of these three parameters and assigns the remaining two from configuration file. The boolean variables *is_ epochs_ varying*, *is_fraction_ samples_ varying*, and *is_ batch_ size_ varying*, set in the configuration file, define which parameter is the varying one.

The choice of the varying parameter is not trivial as it depends on the specific scenario. For instance, involving a enough number of samples in each local update is crucial, thus if the fraction of samples is the varying parameter, devices with small IPS may involve too few samples, risking inadequate updates. Conversely, increasing the batch size leads to higher memory usage, which could be a constraint for devices with limited memory. Additionally, setting too many epochs for devices with high IPS can cause overfitting, especially when few examples are used in the local update.

The parameters are set as follows:

- If the number of epochs is the varying parameter, $batch_size$ and $fraction_samples$ are set according to the static configuration parameters *batch_ size* and *fraction_ samples*, and $epochs_k$ is computed as:

$$epochs_k = \frac{\text{local_iterations}_k \cdot \text{batch_size}}{num_samples_k \cdot \text{fraction_samples}} \tag{4.5}$$

- If the fraction of samples is the varying parameter, *epochs* and $batch_size$ are set according to the static configuration parameters *epochs* and *batch_ size*, and $fraction_samples_k$ is computed as:

$$fraction_samples_k = \frac{\text{local_iterations}_k \cdot \text{batch_size}}{num_samples_k \cdot \text{epochs}} \tag{4.6}$$

- If the batch size is the varying parameter, *epochs* and $fraction_samples$ are set according to the static configuration parameters *epochs* and *fraction_samples*, and $batch_size_k$ is computed as:

$$batch_size_k = \frac{\text{epochs} \cdot num_samples_k \cdot \text{fraction_samples}}{\text{local_iterations}_k} \tag{4.7}$$

This optimizer takes from the *Best RT Optimizer* as it scales parameters depending on the computational resources of selected clients. However, instead of considering the maximum IPS, the computation time is the constant factor here. This distinction is beneficial when the goal is to maximize the number of local iterations within a specified time frame. Moreover, this optimizer provides resource-aware workload allocation not only in the number of epochs, as with the *Best RT Optimizer*, but also in the batch size and the fraction of samples.

The pseudocode of the corresponding `configure_fit` function is shown in Algorithm 4.9.

Algorithm 4.9: ECT optimizer

```
def configure_fit(server_round: int):
    ips_clients = []

    # Retrieve IPS of sampled clients
    for client in clients:
        properties_res = client.get_properties()
        ips = properties_res.properties["ips"]
        ips_clients.append((client, ips))

    # Compute local iterations for each client
    local_iterations = {}
    for client, ips in ips_clients:
        local_iterations[client] = int(comp_time * ips)
    # If epochs is the parameter varying:
    if varying_config["epochs"]:
        batch_size = cfg.batch_size
        fraction_samples = cfg.fraction_samples
        num_samples = samples_client * fraction_samples
```

```
        for client, local_iteration in local_iterations:
            epochs = (local_iteration * batch_size) /
                        num_samples
            config = {
                "local_epochs": local_epochs,
                "batch_size": batch_size,
                "fraction_samples": fraction_samples,
            }
            config[client] = config
    # Analogous in the other two cases
    [...]
    return config
```

5 | Experiments

In this chapter, we present the experiments conducted to verify the performance of the techniques we implemented, as presented in Chapter 4. In Section 5.1, we describe the setup for the experiments. In Section 5.2, we present the results.

5.1. Setup

In this section, we present the setup of the experiments conducted to assess the impact of the strategies and techniques we implemented on the federated learning process.

In federated learning, as in machine learning in general, the correctness of the outcome is not the sole aspect to consider; instead, each modification to the algorithm can impact a variety of metrics, including training accuracy, convergence speed, test accuracy, and loss. Therefore, it is important to set up experiments that allow for the comparison of different techniques under various settings to analyze the strengths and weaknesses of each technique.

5.1.1. Task

As highlighted in Chapter 2, multiple tasks can be addressed by machine learning and federated learning. For our experimentation phase, we opted for image classification due to several reasons:

- The availability of multiple open datasets that facilitate research and experimentation.
- Image classification is one of the most widespread task in federated learning, as demonstrated in [35], [37], and [57].
- The sensitive nature of images makes federated learning particularly attractive for privacy preservation.
- Image classification tasks often involve non-IID data, presenting a realistic challenge for federated learning models.

- Techniques and insights gained from federated image classification models are often transferable to other domains, such as natural language processing, signal processing, and complex tasks like drug discovery and genomics.

To provide more meaningful comparisons, we experimented with two types of networks:

1. A Multi-Layer Perceptron with 2 hidden layers having 64 and 30 neurons, respectively. Dropout is applied after the first hidden layer, and the input is the flattened image.
2. A deep convolutional neural network with 2 convolutional layers followed by max pooling. This is succeeded by 4 fully connected linear layers with 120, 100, 84, and 50 neurons, respectively.

5.1.2. Dataset loading

A crucial aspect of reproducing a federated learning environment is simulating a realistic setting in terms of data distribution. As discussed in Chapter 3, data heterogeneity across clients in federated learning (FL) settings presents a well-recognized challenge. It arises from the varied and uneven distribution of data across participating clients, with each device's data reflecting the unique behavior, preferences, and environment of its user, resulting in non-IID (not independently and identically distributed) datasets.

While several techniques to emulate non-IID datasets in FL have been proposed [40, 47, 60], these often oversimplify the complexity of realistic FL scenarios, which typically involve a larger number of clients and more intricate data distributions.

We adopted the method proposed by Hsu et al. [25], employing the Dirichlet distribution $\mathrm{Dir}_K(\alpha)$ to construct heterogeneous data partitions among clients, with the parameter α controlling the extent of data heterogeneity (imbalance in data size and label distribution among clients). A smaller value of α signifies greater data heterogeneity.

In Figure 5.1 and Figure 5.2, we plot the distribution of samples over the first ten clients, obtained through Dirichlet distribution with parameter α equal to 2 and 0.6, respectively. In different colours, we represent the different classes in each partition. The parameter α influences both the number of samples in each partition and the balance of samples among classes, as shown in the plots.

Figure 5.1: Distribution of samples over clients with alpha=2

Figure 5.2: Distribution of samples over clients with alpha=0.6

To ensure our experiments were comprehensive, we employed two distinct datasets:

1. The MNIST dataset [34], comprising 28x28 pixel images of handwritten digits, with 60,000 images for training and 10,000 for testing, distributed across 10 classes corresponding to the digits 0 through 9. In Figure 5.3, we show the first ten samples of the MNIST dataset, with the corresponding labels.

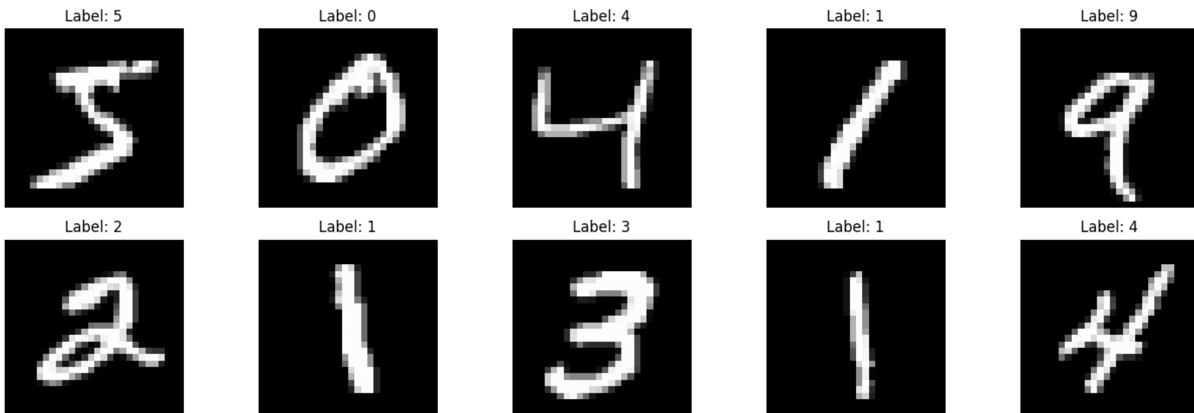

Figure 5.3: First ten samples of the MNIST dataset

2. The CIFAR-10 dataset [33], which includes 60,000 32x32 color images across 10 classes, each class containing 6,000 images. The dataset is split into 50,000 training images and 10,000 test images. In Figure 5.4, we show the first ten samples of the CIFAR-10 dataset, with the corresponding labels.

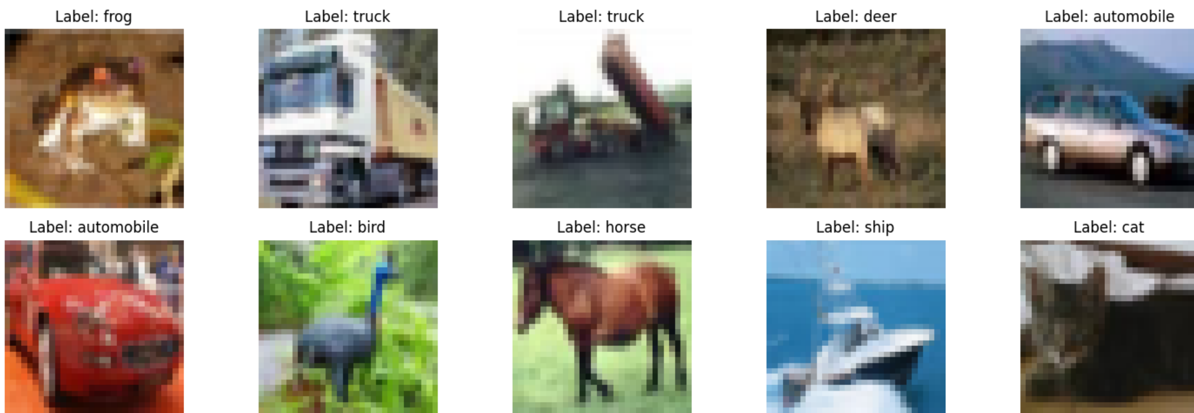

Figure 5.4: First ten samples of the CIFAR-10 dataset

5.1.3. Simulation parameters

All experiments were run using the Flower simulation engine, on a MacOS laptop equipped with 32GB of RAM and an M1 Pro chip. Given that the employed networks were quite simple, all experiments were able to run on the CPU without the need for a GPU. However, the code supports running on a GPU to facilitate the reproduction of experiments with larger networks.

As highlighted in Chapter 3, one advantage of the Flower simulation engine is its ability to simulate experiments with a large number of clients, thanks to its efficient process

management. This feature was exploited to better replicate a federated environment in terms of the number of clients, compared to what would have been achievable by renting physical devices.

This section presents the default setup of the experiments. Specific values for each experiment, if different from the default, are detailed in the respective subsections. Table 5.1 recaps the default values of the parameters common to all experiments.

Parameter	Description	Value
`num_clients`	Total number of clients	100
`num_rounds`	Total number of rounds	200
`epochs`	Default number of epochs on clients	4
`batch_size`	Default batch size on clients	32
`fraction_samples`	Default fraction of samples used by clients	1.0

Table 5.1: Default parameters for the experiments

5.2. Results

In this section, we present the results of the experiments conducted.

5.2.1. Dynamic selection

The first group of experiments compares different techniques of dynamic client selection in federated learning, employing the techniques presented in Chapter 4.

In Table 5.2, we present the values of the parameters for these experiments.

Parameter	Strategy	Description	Value
`C`	*FedAvg*, *pow-d*, *cpow-d*, *rpow-d*	Fraction of clients to select at each round	0.1
`d`	*pow-d*, *cpow-d*	Number of clients to select in the *first_phase*	20
`b`	*cpow-d*	Mini-batch size to use in the *first_phase*	64
C_0	*dyn-sampling*	Initial sampling rate	0.2
`beta`	*dyn-sampling*	Sampling rate decay coefficient	0.1

Table 5.2: Parameters for the experiments on dynamic selection.

Multi Layer Perceptron on MNIST

The setting for these experiments involves an image classification task using the MNIST dataset, partitioned into *num_ clients* partitions by means of a Dirichlet distribution $\mathrm{Dir}_K(\alpha)$ to create heterogeneous data partitions among clients, with the parameter α set to 2 for a balanced dataset, and 0.6 for a highly unbalanced dataset.

These experiments compare the techniques *FedAvg*, *dynamic sampling*, *pow-d*, *cpow-d*, and *rpow-d*.

In Figure 5.5 we plot the test accuracy obtained by performing on the server an evaluation of the global model on the test set, composed of 10000 images in the case of MNIST. The results in this plot are obtained on a dataset partitioned with parameter $\alpha = 2$, so quite balanced.

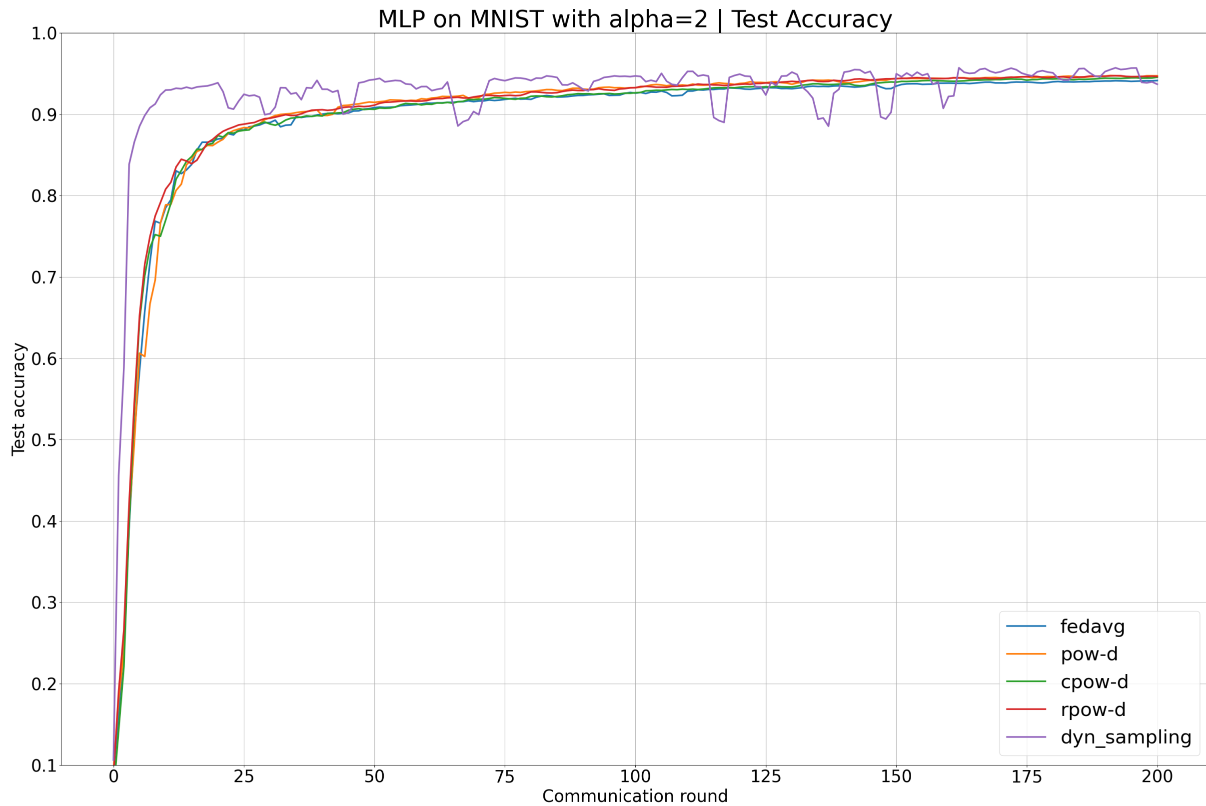

Figure 5.5: MLP on MNIST with α=2, Test accuracy

From the results we can see that *pow-d*, *cpow-d* and *rpow-d* strategies performances are very close to those of *FedAvg*. This is expected because the *Power of Choice* family of strategies give their best in case of highly unbalanced datasets.

The *dynamic sampling* strategy instead presents quite different performances: it converges to a well performing model (in the order of 90% accuracy on the test set) much

quicker that the other strategies, but then presents very unstable performances as the simulation continues, oscillating between 90% and 95% accuracy until round 200. This behavior can be explained by the fact that the dynamic sampling strategy involves a very large number of clients in the first rounds (around 20 in these experiments, double the number of the other strategies) to quickly converge to a good performing model, and then decreases the number of clients selected as the simulation proceeds. In the final rounds of the simulation, this technique randomly selects only 1 client at each round, and that's why it fails to converge to a stable solution and presents very variable performances on the test set.

In Figure 5.6 we plot the test accuracy obtained by performing a server-side evaluation of the global model on the test set. The results in this plot are obtained on a dataset partitioned with parameter $\alpha = 0.6$, so highly unbalanced.

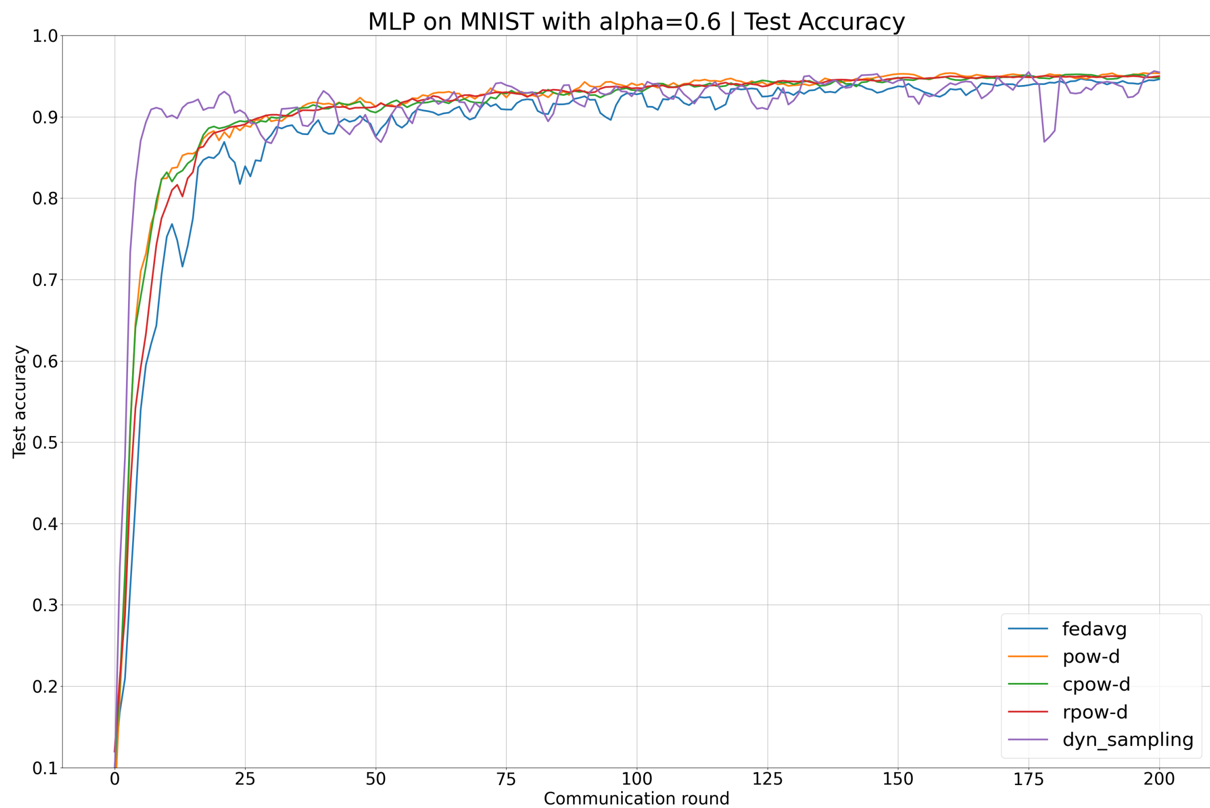

Figure 5.6: MLP on MNIST with α=0.6, Test accuracy

From the plot we see that, as expected, the *pow-d*, *cpow-d* and *rpow-d* strategies perform better than the random sampling *FedAvg* strategy, both in terms of quicker convergence, resulting in better test accuracy, in rounds from 0 to 100, where the blue line (corresponding to FedAvg) always stays below the yellow, green and red lines (corresponding to the Power of Choice strategies), and in terms of stability, given that *FedAvg* presents

highly instability in its test accuracy for the entire duration of the simulation. This result can be explained by considering that random sampling does not take into account any information given from clients while selecting those participating to each round, resulting in the possibility to select clients having very few samples, as for example client 2 in Figure 5.2. The Power of Choice family of strategies, instead, selects clients also based on the number of samples they have locally, having an higher probability of selecting clients with larger number of local samples.

The performances of *dynamic sampling* strategy still present a quicker convergence to a good model with respect to the other strategies, but this converging to a worse model in the long run compared to that of the other techniques, as we can see from the purple line staying below the other lines for almost all the rounds between 25 and 200. This is for the same reasons of *FedAvg*, exacerbated by the fact that *dynamic sampling* diminishes the number of clients sampled at each round.

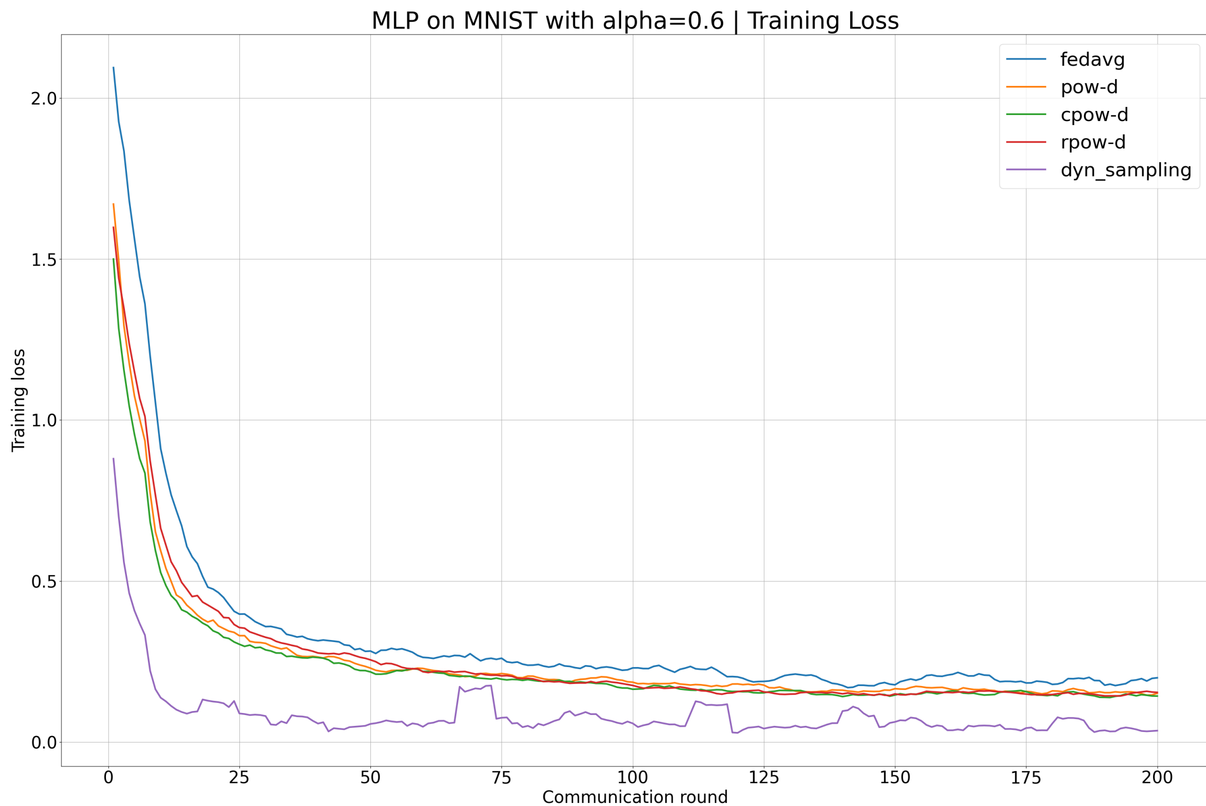

Figure 5.7: MLP on MNIST with α=0.6, Training loss

In Figure 5.7 we plot the training loss of the clients over the simulation, where we can see that dynamic sampling presents a quicker convergence to a good model (lower loss earlier). Such a much quicker convergence can also be explained by the fact that in this strategy we employed an Adam optimizer in the clients, that starts with a high learning

rate in the first rounds and then diminishes it, resulting in faster convergence.

Strategy	*alpha*	**Test Acc.@100**	**Test Acc.@200**	**Train Loss@100**	**Train Loss@200**
FedAvg	2	0.9283	0.9411	0.2695	0.2332
dyn_sampling	2	0.9441	0.9550	0.0754	0.0161
pow-d	2	0.9334	0.9468	0.2381	0.2188
cpow-d	2	0.9241	0.9447	0.2647	0.2301
rpow-d	2	0.9330	0.9484	0.2378	0.1782
FedAvg	0.6	0.9284	0.9429	0.2532	0.2089
dyn_sampling	0.6	0.9123	0.9470	0.0257	0.0432
pow-d	0.6	0.9463	0.9538	0.1804	0.1847
cpow-d	0.6	0.9283	0.9484	0.1493	0.1361
rpow-d	0.6	0.9312	0.9510	0.1560	0.1429

Table 5.3: MLP on MNIST, Test Accuracy and Training Loss values

Convolutional Neural Network on CIFAR10

In these experiments, we repeat the comparisons presented in the previous section under a different setting. In this case, we employ a CNN to classify images from the CIFAR10 dataset. The dataset is partitioned into *num_clients* partitions using a Dirichlet distribution $\text{Dir}_K(\alpha)$ with the parameter α set to 2, to create a fairly balanced dataset, and 0.6, to create a highly unbalanced dataset.

We compare the techniques *FedAvg*, *dynamic sampling*, *pow-d*, *cpow-d*, and *rpow-d*.

Figure 5.8 plots the test accuracy obtained by performing server-side evaluation of the global model on the test set, composed of 10,000 images in the case of CIFAR10. The results in this plot are obtained on a dataset partitioned with parameter $\alpha = 2$, indicating a balanced dataset.

These results confirm those of the previous section, with *FedAvg*, *pow-d*, *cpow-d*, and *rpow-d* performing similarly in the case of a balanced dataset. The performance of *dynamic sampling* is again very unstable, showing quick convergence with higher test accuracy in rounds 0-50, but then becoming unstable and showing worse steady-state performance in rounds 100-200.

In Figure 5.9, we plot the test accuracy obtained by performing server-side evaluation of the global model on the test set. The results in this plot are obtained on a dataset partitioned with parameter $\alpha = 0.6$, indicating a highly unbalanced dataset.

In the presence of an unbalanced dataset, the results are consistent with those obtained

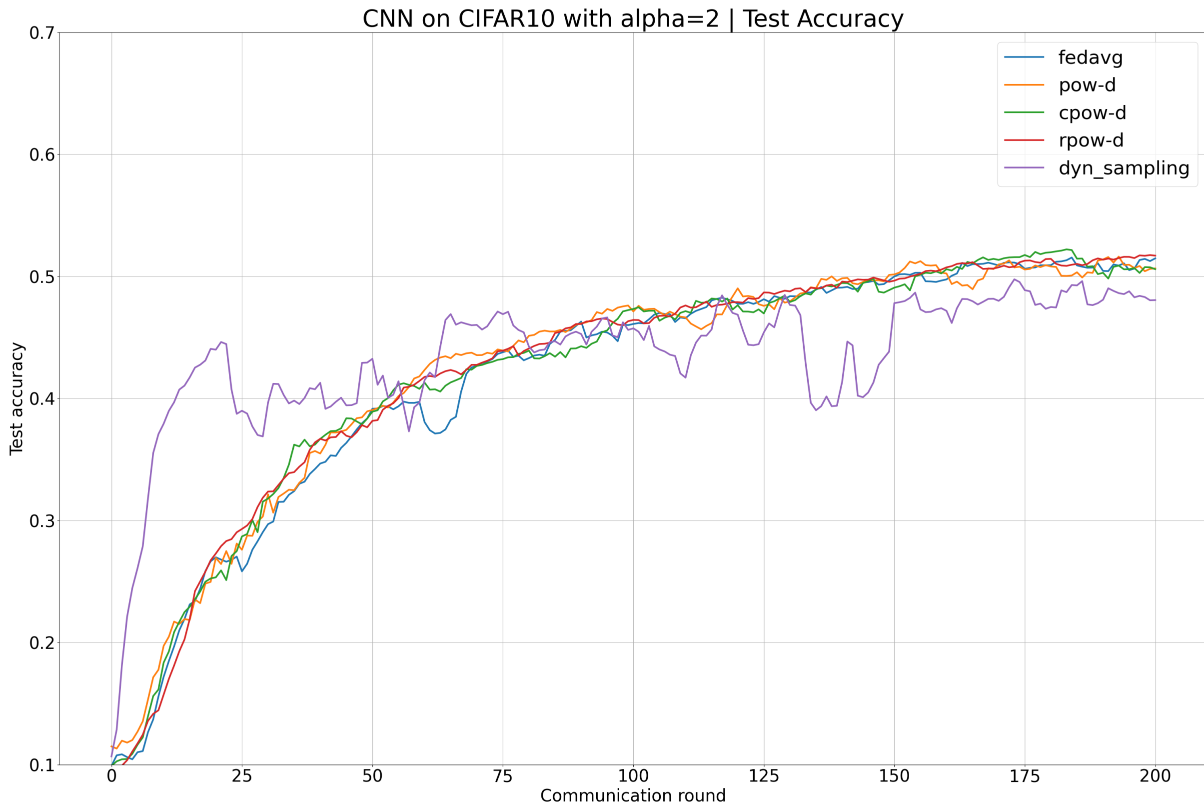

Figure 5.8: CNN on CIFAR10 with α=2, Test accuracy

using a MLP. The *FedAvg* strategy performs worse than the Power of Choice strategies, particularly in the first 125 rounds, as can be seen from the blue line always being below the yellow, red, and green ones. The *dynamic sampling* strategy converges faster but then remains very unstable and performs much worse than the other techniques in the long run.

In this experiment, we also observe that the *rpow-d* strategy takes slightly longer to converge compared to the other strategies. This can be explained by the fact that it saves computation and communication costs by removing the first phase of client evaluation. As a result, it relies on a proxy for the optimal choice, which may not always select the best clients at each round, especially if some well-performing clients have not yet been selected and thus have their loss set to ∞.

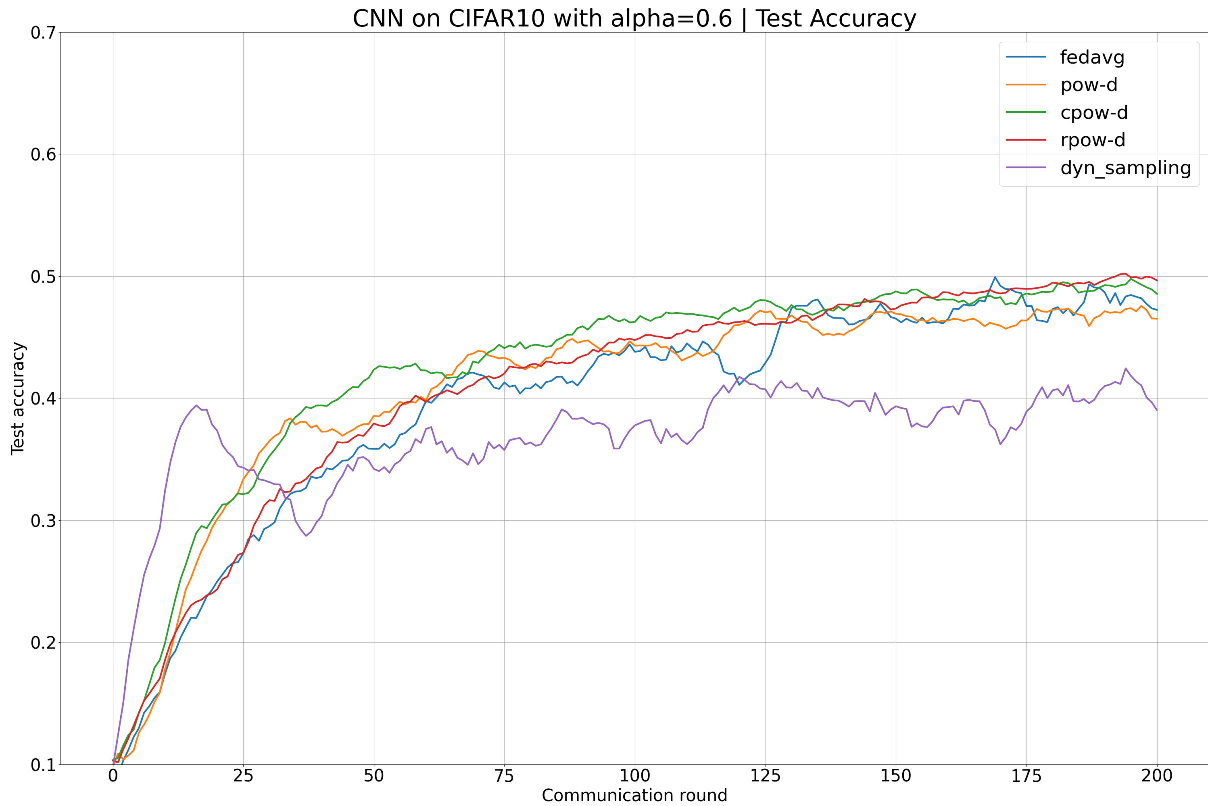

Figure 5.9: CNN on CIFAR10 with α=0.6, Test accuracy

Strategy	*alpha*	Test Acc.@100	Test Acc.@200	Train Loss@100	Train Loss@200
FedAvg	2	0.4608	0.5120	1.2441	0.9622
dyn_sampling	2	0.4584	0.4898	0.4473	0.4034
pow-d	2	0.4774	0.5055	1.2117	1.0238
cpow-d	2	0.4773	0.5065	1.2065	0.9929
rpow-d	2	0.4628	0.5176	1.2169	1.0014
FedAvg	0.6	0.4366	0.4746	0.8496	0.7266
dyn_sampling	0.6	0.3761	0.3944	0.6432	0.5296
pow-d	0.6	0.4426	0.4674	0.8344	0.6666
cpow-d	0.6	0.4633	0.4892	0.8031	0.6500
rpow-d	0.6	0.4455	0.4984	0.9699	0.6837

Table 5.4: CNN on CIFAR10, Test Accuracy and Training Loss values

5.2.2. Resource-aware Workload Selection

This second group of experiments compares the different techniques of resource-aware workload selection presented in Chapter 4.

In Table 5.5, we present the values of the parameters for these experiments.

Parameter	Optimizer	Description	Value
`epochs_min`	*uniform*	Minimum number of epochs	1
`epochs_max`	*uniform*	Maximum number of epochs	5
`batch_size_min`	*uniform*	Minimum batch size	32
`batch_size_max`	*uniform*	Maximum batch size	128
`fraction_samples_min`	*uniform*	Minimum fraction of samples	0.1
`fraction_samples_max`	*uniform*	Maximum fraction of samples	1.0
`epochs`	*ecto*	Default number of epochs	2
`batch_size`	*ecto*	Default batch size	32
`fraction_samples`	*ecto*	Default fraction of samples	1.0
`frac_samples_varying`	*ecto*	Vary or not the fraction of samples	True
`comp_time`	*rt, ecto*	Computation time for each client	30
`mean_ips`	*rt, ecto*	IPS mean value	100
`var_ips`	*rt, ecto*	IPS variance value	50

Table 5.5: Parameters for the experiments on workload selection.

Multi Layer Perceptron on MNIST

In these experiments, we employ a Multi Layer Perceptron to learn an image classification task on the MNIST dataset, partitioned into *num_ clients* partitions using a Dirichlet distribution $\text{Dir}_K(\alpha)$ to create heterogeneous data partitions among clients. The parameter α is set to 0.6 to create a highly unbalanced dataset.

We chose to analyze this setting directly because it provided the most meaningful results in previous experiments and replicates a highly unbalanced dataset, which is common in real-world settings.

These experiments compare the *Global Update Optimizers* implemented, namely *static optimizer*, *uniform optimizer*, *RT optimizer*, and *ECTO optimizer*.

In Figure 5.10, we plot the test accuracy obtained by performing a server-side evaluation of the global model on the test set at the end of each round.

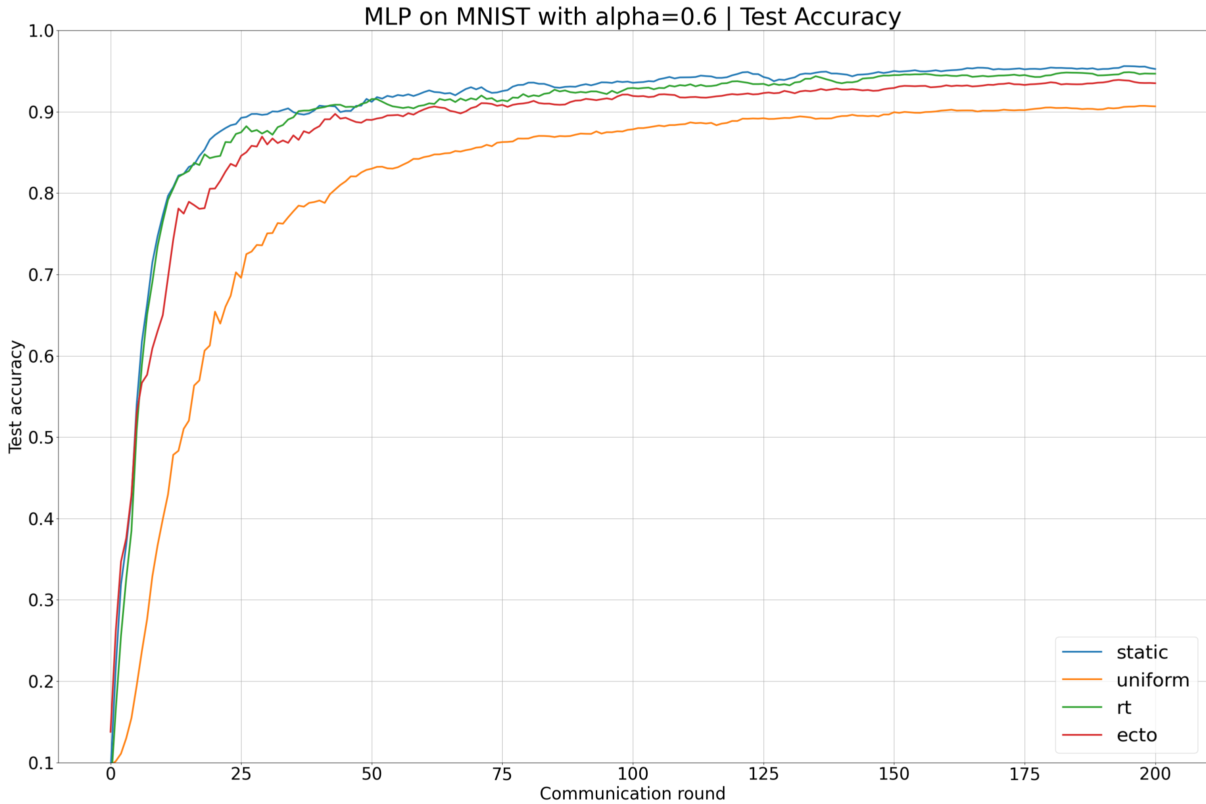

Figure 5.10: Workload optimizers, MLP on MNIST with α=0.6, Test accuracy

From the plot, it is evident that the *static* and *RT* optimizers achieve similar performances, with the former being slightly better. The *ECTO* optimizer follows closely but shows worse results, particularly in the first 50 rounds. The *uniform* optimizer performs the worst, exhibiting over 5% less accuracy than the other techniques after 200 rounds.

These results allow us to infer the following key points:

- The *static* optimizer being the best performing one was expected. This is because it sets the highest possible workload (*epochs* to 4, *batch_size* to 32, and *fraction_samples* to 1.0) for every client at each round, without considering less performing clients. This approach could result in longer computation times in the case of slow clients.

  However, the fact that the RT optimizer, which dynamically sets the number of epochs for each client depending on their computational power, performs closely to the *static* optimizer, suggests it as a very viable alternative. It manages to maintain a fixed computation time per round, regardless of the presence of slower clients.

- The *uniform* optimizer, which assigns workloads by randomly picking parameters from a uniform distribution, performs poorly. This confirms that dynamic workload

assignment requires a logic that takes into account client properties such as computational power or dataset size, especially in cases of unbalanced datasets. Incorrect settings, like the batch size, can prevent clients with few samples from contributing effectively to the model.

These two key takeaways are further confirmed by examining the training loss of the clients over the entire simulation, as plotted in Figure 5.11. This plot also shows the same rank in terms of performance.

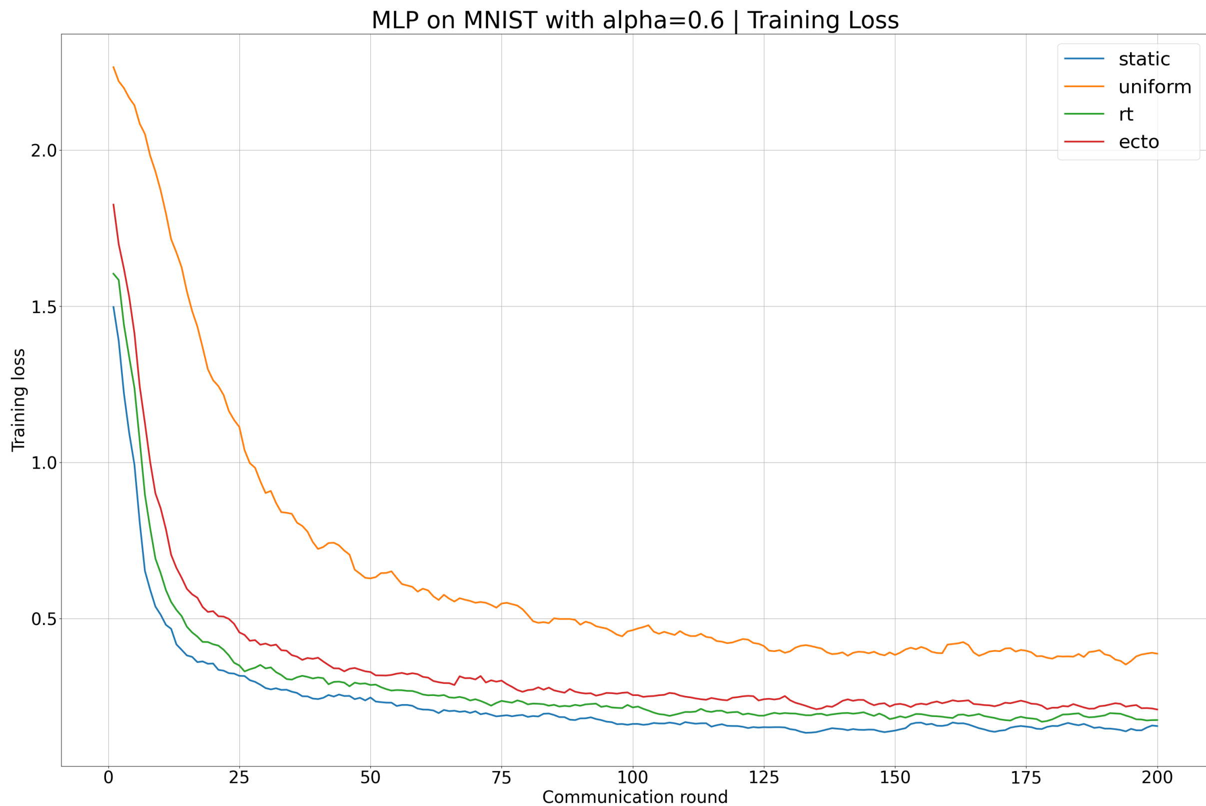

Figure 5.11: Workload optimizers, MLP on MNIST with α=0.6, Training loss

Finally, if we examine the variance of the training loss among clients, as shown in Figure 5.12, it becomes evident that the *uniform* optimizer exhibits the highest variance. This is attributed to the workload parameters being assigned randomly, resulting in a very unstable training throughout the entire duration of the simulation.

The *static* and *RT* optimizers, however, perform similarly in this metric as well. They both start with a very high variance in the initial rounds and gradually converge to zero over the course of 200 rounds.

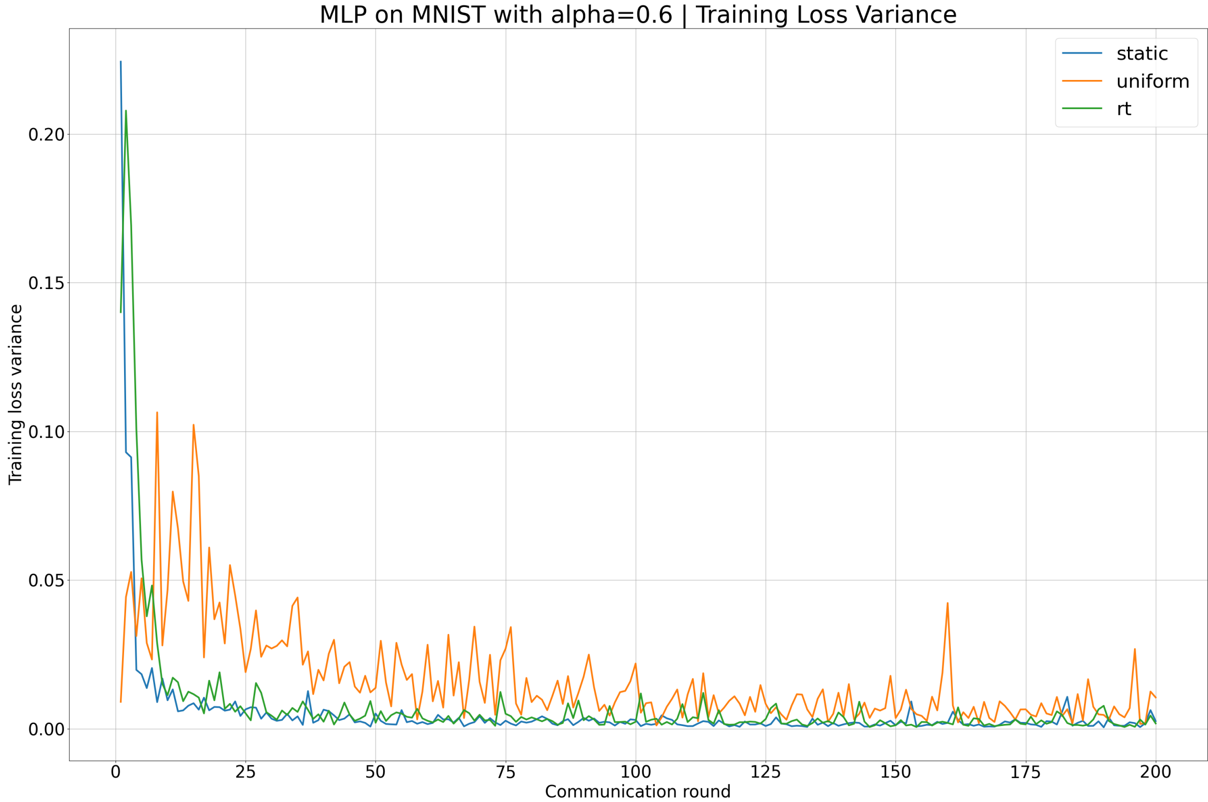

Figure 5.12: Workload optimizers, MLP on MNIST with α=0.6, Loss variance

Strategy	*alpha*	Test Acc.@100	Test Acc.@200	Train Loss@100	Train Loss@200
static	0.6	0.9360	0.9547	0.1617	0.1512
uniform	0.6	0.8760	0.9066	0.4587	0.3795
RT	0.6	0.9265	0.9474	0.2144	0.1747
ECTO	0.6	0.9181	0.9365	0.2571	0.2101

Table 5.6: MLP on MNIST, Test Accuracy & Training Loss with optimizers

Convolutional Neural Network on CIFAR10

In these experiments, we employ a Convolutional Neural Network to learn an image classification task on the CIFAR10 dataset, partitioned into *num_ clients* partitions using a Dirichlet distribution $\mathrm{Dir}_K(\alpha)$ to create heterogeneous data partitions among clients, with the parameter α set to 2.

These experiments compare the *Global Update Optimizers* implemented, namely *static optimizer*, *uniform optimizer*, *RT optimizer*, and *ECTO optimizer*.

In Figure 5.13, we plot the test accuracy obtained by performing a server-side evaluation of the global model on the test set at the end of each round.

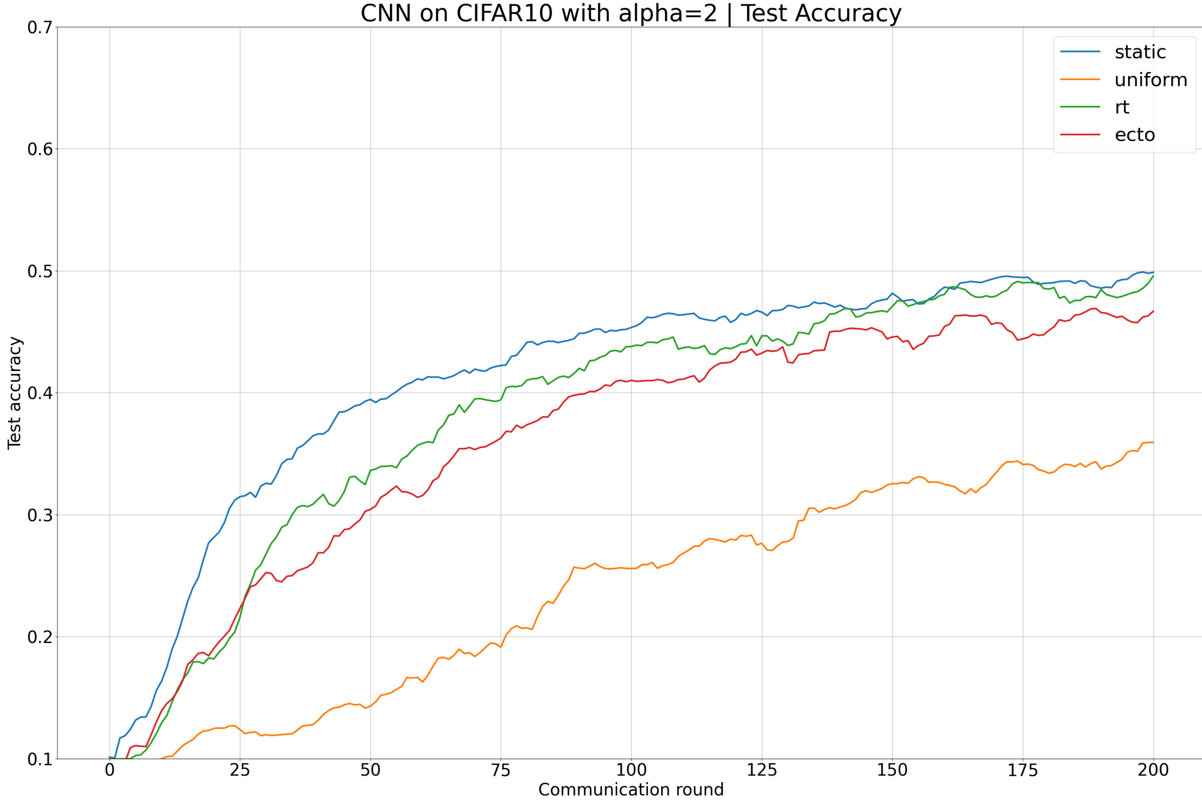

Figure 5.13: Workload optimizers, CNN on CIFAR10 with α=2, Test accuracy

From the results, it is evident that the different optimizers behave similarly to that observed in the previous experiment. The *static* optimizer shows the best performance, closely followed by the *RT* optimizer, which in this case demonstrates slower convergence, and the *ECTO* optimizer. The *uniform* optimizer remains the worst performing, showing even poorer performance compared to the other strategies than those observed in the previous experiment, despite the dataset being more balanced. This may be attributed to the task being somewhat more challenging for the network, as indicated by the overall lower test accuracy in this task.

Similar observations can be made when examining the training loss of the clients over the entire simulation, as shown in Figure 5.14.

5.2.3. Takeaways from the experiments

This section consolidates the key findings from the series of experiments conducted on dynamic client selection and resource-aware workload selection, using both Multi Layer

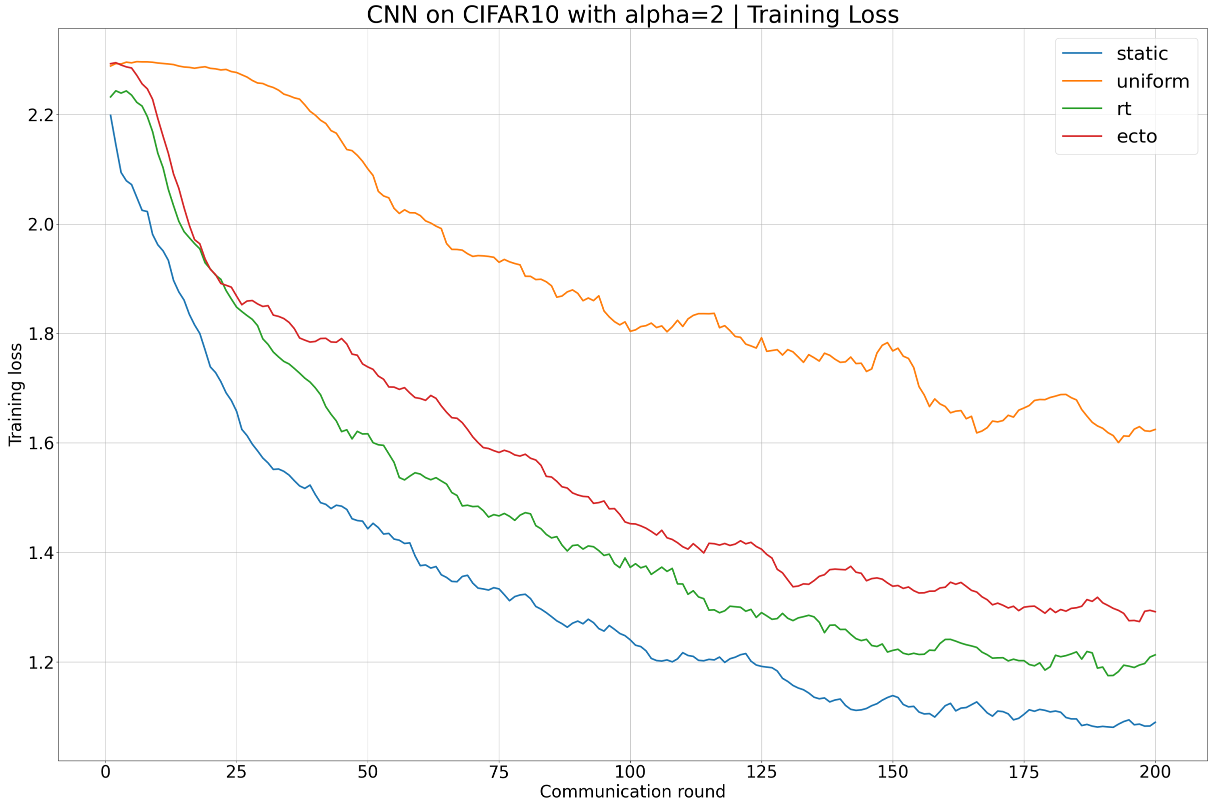

Figure 5.14: Workload optimizers, CNN on CIFAR10 with α=2, Training loss

Strategy	*alpha*	Test Acc.@100	Test Acc.@200	Train Loss@100	Train Loss@200
static	2	0.4519	0.4979	1.2403	1.0898
uniform	2	0.2559	0.3592	1.8038	1.6246
RT	2	0.4376	0.4969	1.3727	1.2130
ECTO	2	0.4090	0.4629	1.4525	1.2917

Table 5.7: CNN on CIFAR10, Test Accuracy & Training Loss with optimizers

Perceptron on MNIST and Convolutional Neural Network on CIFAR10.

The experiments on dynamic client selection revealed several insights:

- The *Power of Choice* strategies (*pow-d*, *cpow-d*, *rpow-d*) demonstrated effectiveness, particularly in unbalanced datasets, outperforming the *FedAvg* in terms of both convergence speed and stability.
- The *dynamic sampling* strategy, while converging quickly, showed instability in performance over time, especially in later rounds of the simulation. This was attributed to its approach of initially involving a large number of clients and gradually reducing this number, affecting long-term model stability.

Key observations from the resource-aware workload selection experiments include:

- The *static* and *RT* optimizers performed consistently well across different settings, indicating their robustness. The *static* optimizer, setting the highest workload, generally led in performance, but the *RT* optimizer's ability to adapt the workload to client capabilities made it a close and more efficient competitor.
- The *uniform* optimizer's random approach to workload distribution resulted in poorer performance, highlighting the necessity for strategies that consider client-specific characteristics like computational power and dataset size.
- Variance analysis of the training loss among clients further confirmed these findings, with the *uniform* optimizer showing the highest variance due to its random workload assignment.

These experiments underscore the importance of strategic client selection and workload distribution in federated learning to achieve effective and stable model training, especially in heterogeneous and real-world scenarios.

6 | Conclusions

This thesis proposes extensions to Flower, a highly promising framework for advancing federated learning research.

The platform offers several advantages:

- A comprehensive set of classes and interfaces that allow easy extension and integration of new strategies and techniques.
- Very well-maintained documentation and an active community to support the development of the project.
- An easy-to-use simulation engine that allows replication of federated learning environments close to reality, enabling easy reproducibility of experiments and prototyping.

Building on this platform, our objective was to address existing gaps in federated learning research, specifically focusing on crucial open issues such as dynamic client selection during training and workload allocation considering client properties and resources.

To this end, we presented four state-of-the-art strategies for dynamic client selection, extended Flower to accommodate them, and compared them against the *FedAvg* baseline.

Subsequently, we proposed four innovative strategies for resource-aware workload allocation, extended Flower to support them and integrated them with the dynamic client selection techniques.

We finally experimented with the implemented techniques by utilizing the built-in Flower simulator. We simulated a federated learning setting with 100 clients over 200 rounds and compared the implemented techniques.

Our experiments revealed the competitiveness of the proposed strategies with state-of-the-art techniques, showcasing superior performance in certain settings, such as heterogeneous clients. In detail:

- The *Power of Choice* strategies exhibited effectiveness, particularly in unbalanced

datasets, outperforming *FedAvg* in terms of convergence speed and stability.

- The *dynamic sampling* strategy, while converging quickly, displayed performance instability over time, attributed to its initial involvement of a large number of clients.
- The *static* and *RT* optimizers consistently performed well across diverse settings, with the latter's adaptive workload distribution proving to be a close and efficient competitor.
- The *uniform* optimizer, relying on a random approach to workload distribution, yielded poorer performance, emphasizing the necessity for strategies considering client-specific characteristics in federated learning scenarios.

In summary, our experiments underscored the significance of strategic client selection and workload distribution in federated learning, particularly in heterogeneous and real-world scenarios, to achieve effective and stable model training.

Part of our work is currently in the process of being merged into the official repository of Flower, which boasts over 3300 stars on GitHub. This contribution positions us as part of the next generation of research in federated learning.

6.1. Future work

Even though the work considered multiple facets of federated learning, it could be extended in various directions:

- Further investigate dynamic selection and resource-aware workload allocation by proposing innovative strategies that take into account multiple client properties, such as battery life, signal level, or network speed. A more comprehensive strategy, considering various properties other than just computational power, would enhance adaptability to diverse scenarios.
- Experiment with other tasks, such as natural language processing using large language models. Recent developments in complex models like GPT [8] and LLaMA [48] have sparked research interest in federated learning for these models [9, 29, 59]. Experimenting with such models involves setting up more powerful clients and enabling parallelization on GPUs due to their high computational demands. While Flower supports this, potential bottlenecks and performance issues need consideration.
- Experiment with a real federated environment: despite highlighting the advantages of Flower's simulation engine, future work could involve setting up a real federated environment with multiple actual devices. Initial investigations revealed that Flower

provides an APK to install a federated learning client on an Android device. Future work could involve renting real mobile phones or using cloud instances like AWS EC2 to deploy clients on actual federated devices. These experiments may reveal nuances not evident in a simulated environment, such as clients slowing down training due to lower computational power. However, reproducing a highly heterogeneous setting, as a real federated environment should be, currently lacks off-the-shelf solutions. Thus, investigations into techniques and tools are necessary.

Bibliography

[1] Multi layer perceptron. URL https://scikit-learn.org/stable/modules/neural_networks_supervised.html.

[2] *Mean Squared Error*, pages 337–339. Springer New York, New York, NY, 2008. ISBN 978-0-387-32833-1. doi: 10.1007/978-0-387-32833-1_251. URL https://doi.org/10.1007/978-0-387-32833-1_251.

[3] L. Baresi, G. Quattrocchi, and N. Rasi. Federated machine learning as a self-adaptive problem. In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 41–47, 2021. doi: 10.1109/SEAMS51251.2021.00016.

[4] L. Baresi, G. Quattrocchi, and N. Rasi. Open challenges in federated machine learning. *IEEE Internet Computing*, 27(2):20–27, 2023. doi: 10.1109/MIC.2022.3190552.

[5] D. J. Beutel, T. Topal, A. Mathur, X. Qiu, T. Parcollet, and N. D. Lane. Flower: A friendly federated learning research framework. *CoRR*, abs/2007.14390, 2020. URL https://arxiv.org/abs/2007.14390.

[6] K. A. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečný, S. Mazzocchi, H. B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roselander. Towards federated learning at scale: System design. *CoRR*, abs/1902.01046, 2019. URL http://arxiv.org/abs/1902.01046.

[7] L. Bottou. *Stochastic Gradient Descent Tricks*, pages 421–436. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35289-8. doi: 10.1007/978-3-642-35289-8_25. URL https://doi.org/10.1007/978-3-642-35289-8_25.

[8] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020. URL https://arxiv.org/abs/2005.14165.

[9] C. Chen, X. Feng, J. Zhou, J. Yin, and X. Zheng. Federated large language model: A position paper, 2023.

[10] F. Chen, Z. Dong, Z. Li, and X. He. Federated meta-learning for recommendation. *CoRR*, abs/1802.07876, 2018. URL http://arxiv.org/abs/1802.07876.

[11] L. Condat, I. Agarský, G. Malinovsky, and P. Richtárik. Tamuna: Doubly accelerated federated learning with local training, compression, and partial participation, 2023.

[12] X. Dai, I. Spasić, B. Meyer, S. Chapman, and F. Andres. Machine learning on mobile: An on-device inference app for skin cancer detection. In *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 301–305, 2019. doi: 10.1109/FMEC.2019.8795362.

[13] FedML.ai. FedML on GitHub. URL https://github.com/FedML-AI/FedML.

[14] Flower authors. Flower monthly. URL https://flower.dev/conf/flower-monthly/.

[15] Flower contributors. Flower Android Example (TensorFlowLite). https://github.com/adap/flower/tree/main/examples/android.

[16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[17] Google. FedJAX on GitHub, . URL https://github.com/google/fedjax.

[18] Google. Descending into ml: Training and loss. https://developers.google.com/machine-learning/crash-course/descending-into-ml/training-and-loss, .

[19] Google. JAX Quickstart, . URL https://jax.readthedocs.io/en/latest/notebooks/quickstart.html.

[20] Google. Tensorflow Federated, . URL https://www.tensorflow.org/federated.

[21] Google. Tensorflow Federated on GitHub, . URL https://github.com/tensorflow/federated.

[22] A. Hard, C. M. Kiddon, D. Ramage, F. Beaufays, H. Eichner, K. Rao, R. Mathews, and S. Augenstein. Federated learning for mobile keyboard prediction, 2018. URL https://arxiv.org/abs/1811.03604.

[23] C. He, S. Li, J. So, M. Zhang, H. Wang, X. Wang, P. Vepakomma, A. Singh, H. Qiu, L. Shen, P. Zhao, Y. Kang, Y. Liu, R. Raskar, Q. Yang, M. Annavaram, and S. Aves-

timehr. Fedml: A research library and benchmark for federated machine learning. *CoRR*, abs/2007.13518, 2020. URL `https://arxiv.org/abs/2007.13518`.

[24] S. Horváth, S. Laskaridis, M. Almeida, I. Leontiadis, S. Venieris, and N. D. Lane. FjORD: Fair and accurate federated learning under heterogeneous targets with ordered dropout. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021. URL `https://openreview.net/forum?id=4fLr7H5D_eT`.

[25] T. H. Hsu, H. Qi, and M. Brown. Measuring the effects of non-identical data distribution for federated visual classification. *CoRR*, abs/1909.06335, 2019. URL `http://arxiv.org/abs/1909.06335`.

[26] Hydra. Hydra library documentation. URL `https://hydra.cc/docs/intro/`.

[27] Y. Jee Cho, J. Wang, and G. Joshi. Towards understanding biased client selection in federated learning. In G. Camps-Valls, F. J. R. Ruiz, and I. Valera, editors, *Proceedings of The 25th International Conference on Artificial Intelligence and Statistics*, volume 151 of *Proceedings of Machine Learning Research*, pages 10351–10375. PMLR, 28–30 Mar 2022. URL `https://proceedings.mlr.press/v151/jee-cho22a.html`.

[28] S. Ji, W. Jiang, A. Walid, and X. Li. Dynamic sampling and selective masking for communication-efficient federated learning. *CoRR*, abs/2003.09603, 2020. URL `https://arxiv.org/abs/2003.09603`.

[29] J. Jiang, X. Liu, and C. Fan. Low-parameter federated learning with large language models, 2023.

[30] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, R. G. L. D'Oliveira, H. Eichner, S. E. Rouayheb, D. Evans, J. Gardner, Z. Garrett, A. Gascón, B. Ghazi, P. B. Gibbons, M. Gruteser, Z. Harchaoui, C. He, L. He, Z. Huo, B. Hutchinson, J. Hsu, M. Jaggi, T. Javidi, G. Joshi, M. Khodak, J. Konečný, A. Korolova, F. Koushanfar, S. Koyejo, T. Lepoint, Y. Liu, P. Mittal, M. Mohri, R. Nock, A. Özgür, R. Pagh, M. Raykova, H. Qi, D. Ramage, R. Raskar, D. Song, W. Song, S. U. Stich, Z. Sun, A. T. Suresh, F. Tramèr, P. Vepakomma, J. Wang, L. Xiong, Z. Xu, Q. Yang, F. X. Yu, H. Yu, and S. Zhao. Advances and open problems in federated learning. 2021. doi: https://doi.org/10.48550/arXiv.1912.04977.

[31] J. Kim, G. Kim, and B. Han. Multi-level branched regularization for federated learning. In K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, vol-

ume 162 of *Proceedings of Machine Learning Research*, pages 11058–11073. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/kim22a.html.

[32] C. Korkmaz, H. E. Kocas, A. Uysal, A. Masry, O. Ozkasap, and B. Akgun. Chain fl: Decentralized federated machine learning via blockchain. In *2020 Second International Conference on Blockchain Computing and Applications (BCCA)*, pages 140–146, 2020. doi: 10.1109/BCCA50787.2020.9274451.

[33] A. Krizhevsky. Learning multiple layers of features from tiny images. 2009. URL https://api.semanticscholar.org/CorpusID:18268744.

[34] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998. ISSN 1558-2256. doi: 10.1109/5.726791.

[35] L. Li, N. Xie, and S. Yuan. A federated learning framework for breast cancer histopathological image classification. *Electronics*, 11(22), 2022. ISSN 2079-9292. URL https://www.mdpi.com/2079-9292/11/22/3767.

[36] Q. Li, B. He, and D. Song. Model-contrastive federated learning. *CoRR*, abs/2103.16257, 2021. URL https://arxiv.org/abs/2103.16257.

[37] B. Liu, B. Yan, Y. Zhou, Y. Yang, and Y. Zhang. Experiments of federated learning for covid-19 chest x-ray images, 2020.

[38] J. Liu, J. Huang, Y. Zhou, X. Li, S. Ji, H. Xiong, and D. Dou. From distributed machine learning to federated learning: A survey. *Knowledge and Information Systems*, 64(4):885–917, 2022. doi: 10.1007/s10115-022-01664-x.

[39] Y. Liu, T. Fan, T. Chen, Q. Xu, and Q. Yang. Fate: An industrial grade platform for collaborative learning with data protection. *Journal of Machine Learning Research*, 22(226):1–6, 2021. URL http://jmlr.org/papers/v22/20-815.html.

[40] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y. Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In A. Singh and J. Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 1273–1282. PMLR, 20–22 Apr 2017. URL https://proceedings.mlr.press/v54/mcmahan17a.html.

[41] OpenMined. PySyft on GitHub. URL https://github.com/OpenMined/PySyft.

[42] D. W. Otter, J. R. Medina, and J. K. Kalita. A survey of the usages of deep learn-

ing for natural language processing. *IEEE Transactions on Neural Networks and Learning Systems*, 32(2):604–624, 2021. doi: 10.1109/TNNLS.2020.2979670.

[43] A. Pacheco, E. Flores, R. Sánchez, and S. Almanza-García. Smart classrooms aided by deep neural networks inference on mobile devices. In *2018 IEEE International Conference on Electro/Information Technology (EIT)*, pages 0605–0609, 2018. doi: 10.1109/EIT.2018.8500260.

[44] N. Rieke. What is federated learning?, Sep 2022. URL `https://blogs.nvidia.com/blog/2019/10/13/what-is-federated-learning/`.

[45] J. H. Ro, A. T. Suresh, and K. Wu. Fedjax: Federated learning simulation with JAX. *CoRR*, abs/2108.02117, 2021. URL `https://arxiv.org/abs/2108.02117`.

[46] A. K. Sahu, T. Li, M. Sanjabi, M. Zaheer, A. Talwalkar, and V. Smith. On the convergence of federated optimization in heterogeneous networks. *CoRR*, abs/1812.06127, 2018. URL `http://arxiv.org/abs/1812.06127`.

[47] F. Sattler, S. Wiedemann, K. Müller, and W. Samek. Robust and communication-efficient federated learning from non-iid data. *CoRR*, abs/1903.02891, 2019. URL `http://arxiv.org/abs/1903.02891`.

[48] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. Llama: Open and efficient foundation language models, 2023.

[49] A. Tyurin and P. Richtárik. DASHA: Distributed nonconvex optimization with communication compression and optimal oracle complexity. In *The Eleventh International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=VA1YpcNr7ul`.

[50] A. Voulodimos, N. Doulamis, A. Doulamis, and E. Protopapadakis. Deep learning for computer vision: A brief review. *Computational Intelligence and Neuroscience*, Feb 2018. URL `https://www.hindawi.com/journals/cin/2018/7068349/`.

[51] J. Wang, Q. Liu, H. Liang, G. Joshi, and H. V. Poor. Tackling the objective inconsistency problem in heterogeneous federated optimization. *CoRR*, abs/2007.07481, 2020. URL `https://arxiv.org/abs/2007.07481`.

[52] Wikipedia contributors. Convolutional neural network — Wikipedia, the free encyclopedia, 2023. URL `https://en.wikipedia.org/w/index.php?title=Convolutional_neural_network`.

[53] Wikipedia contributors. Cross-entropy — Wikipedia, the free encyclopedia, 2023. URL https://en.wikipedia.org/w/index.php?title=Cross-entropy.

[54] Wikipedia contributors. Mnist database — Wikipedia, the free encyclopedia, 2023. URL https://en.wikipedia.org/w/index.php?title=MNIST_database&oldid=1174503124.

[55] Wikipedia contributors. Mean squared error — Wikipedia, the free encyclopedia, 2023. URL https://en.wikipedia.org/w/index.php?title=Mean_squared_error.

[56] Wikipedia contributors. Stochastic gradient descent — Wikipedia, the free encyclopedia, 2023. URL https://en.wikipedia.org/w/index.php?title=Stochastic_gradient_descent&oldid=1180465768.

[57] D. Ye, R. Yu, M. Pan, and Z. Han. Federated learning in vehicular edge computing: A selective model aggregation approach. *IEEE Access*, 8:23920–23935, 2020. doi: 10.1109/ACCESS.2020.2968399.

[58] P. Yu, L. Wynter, and S. Lim. Fed+: A family of fusion algorithms for federated learning. 09 2020.

[59] Z. Zhang, Y. Yang, Y. Dai, L. Qu, and Z. Xu. When federated learning meets pre-trained language models' parameter-efficient tuning methods, 2023.

[60] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra. Federated learning with non-iid data. *CoRR*, abs/1806.00582, 2018. URL http://arxiv.org/abs/1806.00582.

[61] A. Ziller, A. Trask, A. Lopardo, B. Szymkow, B. Wagner, E. Bluemke, J.-M. Nounahon, J. Passerat-Palmbach, K. Prakash, N. Rose, T. Ryffel, Z. N. Reza, and G. Kaissis. *PySyft: A Library for Easy Federated Learning*, pages 111–139. Springer International Publishing, Cham, 2021. ISBN 978-3-030-70604-3. doi: 10.1007/978-3-030-70604-3_5. URL https://doi.org/10.1007/978-3-030-70604-3_5.

List of Figures

2.1 The architecture of a neuron in a DNN. 9
2.2 A MLP with one output neuron. 11
2.3 The overall architecture of a CNN. 12
2.4 Comparison between the three approaches. 14

3.1 Federated learning applied to the healthcare sector. 15
3.2 Flower server class diagram. 31
3.3 Flower core framework architecture. 34

4.1 The five strategies implemented. 38
4.2 The four workload optimizers implemented 49

5.1 Distribution of samples over clients with alpha=2 59
5.2 Distribution of samples over clients with alpha=0.6 59
5.3 First ten samples of the MNIST dataset 60
5.4 First ten samples of the CIFAR-10 dataset 60
5.5 MLP on MNIST with α=2, Test accuracy 62
5.6 MLP on MNIST with α=0.6, Test accuracy 63
5.7 MLP on MNIST with α=0.6, Training loss 64
5.8 CNN on CIFAR10 with α=2, Test accuracy 66
5.9 CNN on CIFAR10 with α=0.6, Test accuracy 67
5.10 Workload optimizers, MLP on MNIST with α=0.6, Test accuracy 69
5.11 Workload optimizers, MLP on MNIST with α=0.6, Training loss 70
5.12 Workload optimizers, MLP on MNIST with α=0.6, Loss variance 71
5.13 Workload optimizers, CNN on CIFAR10 with α=2, Test accuracy 72
5.14 Workload optimizers, CNN on CIFAR10 with α=2, Training loss 73

List of Tables

3.1 Strengths and weaknesses of FL frameworks. 29

4.1 Parameters of the FedAvg technique 38
4.2 Parameters of the Dynamic Sampling technique 41
4.3 Parameters of the Power of Choice techniques 43

5.1 Default parameters for the experiments 61
5.2 Parameters for the experiments on dynamic selection. 61
5.3 MLP on MNIST, Test Accuracy and Training Loss values 65
5.4 CNN on CIFAR10, Test Accuracy and Training Loss values 67
5.5 Parameters for the experiments on workload selection. 68
5.6 MLP on MNIST, Test Accuracy & Training Loss with optimizers 71
5.7 CNN on CIFAR10, Test Accuracy & Training Loss with optimizers 73

Acknowledgements

Thanks to Professor Luciano Baresi and Tommaso Dolci for their invaluable support throughout the entire duration of the work.