POLITECNICO DI TORINO

Master's Degree in Data Science and Engineering

Master's Degree Thesis

Speeding up convergence while preserving privacy in Heterogeneous Federated Learning

Supervisors

Prof. Barbara CAPUTO

Dr. Marco CICCONE

Dr. Debora CALDAROLA

Candidate

Andrea RIZZARDI

December 2022

Acknowledgements

Un ringraziamento di dovere va ai Professori che mi hanno accompagnato in questi ultimi due anni. Ringrazio in particolare la mia relatrice Barbara Caputo ed i miei correlatori Marco e Debora per aver affrontato insieme a me questo lavoro.
Ringrazio i miei amici di una vita, che anche in questo importante passo ci sono stati.
Ringrazio le nuove amicizie fatte lungo questo percorso, con la speranza che, in un futuro, possano rientrare nei ringraziamenti precedenti.
Ringrazio la mia famiglia, per avermi sostenuto anche questa volta.
Ringrazio Torino, per avermi ospitato ed accolto in questi ultimi due anni.
Ed infine ringrazio il me del passato, che con i suoi sforzi è riuscito a farmi arrivare a scrivere questi ringraziamenti con la consapevolezza di aver concluso un percorso incredibile.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
artificial intelligence

ML
machine learning

SL
supervised learning

GDPR
General Data Protection Regulation

FL
Federated Learning

FMTL
Federated MultiTask Learning

CNN
Convolutional Neural Network

RNN
Recurrent Neural Network

SOTA
State Of The Art

PCA
Principal Components Analysis

GAN

Generative Adversarial Network

NCP

Next Character Prediction

LSTM

Long Short Term Memory

FID

Fréchet inception distance

Chapter 1

Introduction

More and more, every day, machine learning and deep learning algorithms are adopted as endeavor to automatize or facilitate many aspect of human life. Thanks to their ability of 'learning' from data, they can be applied in different contexts: in the image domain for instance, the object recognition task[1] is one of the major examples of how useful a ML algorithm can be with applications in the self driving cars or robot industries. In the same domain, other interesting applications of ML algorithms can be found in the medical sector, where, for example, the goal is to recognize a tumor from an image [2].
The term *learning* has been adopted to emphasize the fact that ML algorithms learn from data. Citing again the tumor recognition task, in order to have an accurate ML algorithm capable of recognize a tumor with a fair confidence, a set of images coupled with the label 'tumor' and another set of images with the label 'no tumor' must be provided. In other words, real-life example are necessary to ML algorithms. Formally this 'learning from examples' approach is denoted as *supervised learning* (SL)[3] in relation to the fact that there must be a manual supervision that 'feeds' the ML algorithm with real-life examples.
Although *SL* approach has been proven to be effective in different tasks from an accuracy point of view, it creates different challenges from the 'data gathering' point of view. Especially if data are related to personal informations of individuals, privacy constraints introduced by very strict international legislations like the European GDPR [4], can represent an obstacle for a ML project.
Let's make a crystal clear example of this problem: let's suppose that we want to create a ML model that is capable of, given an image of a face, recognize the age of the person in the image. In order to train this model, we need to gather locally thousands of images of faces, coupled with the age of each person. Clearly, we are obtaining personal informations of the people in the images set. To ask a formal permission to each person we have a photo of, it can represent a logistical challenge for itself, since, usually, modern ML models requires very big datasets.

The Federated Learning (FL) paradigm is born exactly as the attempt to solve this problem [5]. The core idea of FL can be summed up in this sentence: instead of 'moving' data from users to the ML algorithm, let's move the ML algorithm to the users. With this approach, the 'gather data' phase is no more necessary, since it is not necessary in FL to collect all the data inside a single local dataset.
The first FL algorithm introduced was FedAvg[5]. It was also the first work that introduced FL as a new ML paradigm, so it set up the basis in FL. In FedAvg there are two actors involved: the central server and the clients. The central server has the role of coordinator, while the clients are all the devices that have private data on which the ML algorithm will be trained on.
A traditional FL train of a model is an iterative process, composed by many rounds. At each round, the central server selects a fraction of clients. To these selected clients the server sends a not-yet-trained model. Each selected client trains locally, on its own data, the model, then it sends back the trained model to the server. As last step, the server combines all the trained models incoming from the selected clients by averaging their weights. The resulting averaged model will be used in the next round.
Even though FL paradigm can be used as a solution to solve the privacy-related problems that supervised learning does not consider, it presents other different challenges related to the decentralized setup: first of all, it is important to remember that the local data distribution is not equal for all the clients, nor in terms of number of examples neither in terms of class distribution. Let's consider a scenario in which the FL paradigm is applied in order to train a ML model capable of recognize the landscape of a photo (mountains, sea, city, ...). The clients will be different mobile phones scattered around the world. It is very plausible that clients geographically located in a coastal country will have many 'sea-labelled' images with respect to clients located in cold countries. Moreover, young-age people tend to use more often the phone, so probably they will have many more photo with respect to the phone of an elder person.
This problem is formally defined as *heterogeneity* problem and it can represent a huge threat for FL model performances [6].
Another FL problem is the fact that all the power computation required in order to train a model is split among different heterogeneous devices, with different computational power and a not-always reliable connection to the central server. This problem is formally called *client reliability* problem [6]. Thinking again at mobile phones as clients, it is very plausible that many of these devices can be old and outdated, while others can be the current state of the art phones. Maybe some of these phones are shut down by the owners, hence it is not possible to maintain the connection with the center server.
The main goal of this work is introducing FedSeq [7], a novel algorithm specifically tuned to ease the struggle of standard FL algorithms, like FedAvg, in heterogeneous

scenarios. FedSeq is based on the following concepts: first of all, all the clients must be grouped together in different clusters. The important rule to follow in order to create the clusters is that, clients with different data distributions must be grouped together in order to cover all the classes inside each group. With this approach, each cluster has an homogeneous data distribution, hence the *heterogeneity* problem will not come up. Remembering the landscape recognition example, with this approach each cluster will be composed by clients with only photo of mountains, clients with only photo of the sea, clients with only photo of cities and so on. From a cluster point of view, the overall images set (considered as the union of the images of all the clients inside the cluster) will cover all the classes (mountains, sea, city, ...). As second step, FedSeq considers, as clients, the aforementioned clusters and then proceeds with the standard FedAvg approach. Specifically, the clients inside a cluster will be form a "chain", and the incoming model from the central server will be received from the first client, trained locally and then directly sent to the next client in the chain. The process will continue until the last client of the chain that will send the model back to the central server. With this approach, the model is also able to 'see' more example for each class per round (guaranteeing better performances), while, in FedAvg, a model is only able to 'see' the examples provided by the client on which is trained on.
The novel FedSeq approach presents by itself new challenges: how is it possible to create clusters based on the local clients data distributions if these informations should be privacy protected? The fact that, inside a cluster, the model is sent directly client-to-client, can create private informations leaks of any kind? How the FedSeq approach responds to malicious clients? In this thesis, all these questions will be deeply analyzed, along with empirical experiments and comparisons with other state of the art FL methods.

Chapter 2

Background

2.1 Machine and Deep Learning

When we think about the industrial revolution, different images come to our minds. Some examples of them might be the steam power, big machines, industrial facilities. They all serve for one purpose and one purpose only: automation. We remember the industrial revolution as the moment in time from which the manual work can be automated by machines. Of course the jobs that can be automated at that time were only manual and repetitive jobs in which there were no decision making process.
In our modern times, we are witnessing a similar phenomena: backed by the support of computers and microelectronic in general, machine learning is gradually transforming our society just like the steam power during the industrial revolution. The difference stands in what is automated. In this new modern revolution that we are witnessing, the decision making process itself is automated and the "machine learning" is the instrument to accomplish that.
When we are talking about machine learning, we are talking about algorithms that learn from real life data a particular task. This task can span from recognizing a particular entity in an image or forecasting the stock price of some company or generating a natural language text.
Depending on the task, the algorithm as to differ in order to achieve good result on it. For instance, if the task is entity recognition, the best algorithm to be used is a Convolutional Neural Network [8] while, for a stock price forecasting task, maybe a simple polynomial regression model could do the job.
As a subset of the whole ML algorithm world, the Deep Learning is defined [9]. A Deep Learning model has to have the following feature: given an input data of any kind, the algorithm has to "map" it in a latent space. Essentially, given an input, the Deep Learning algorithm has to produce a vector representation of it. On of

the first work in this regard is Word2Vec [10], a Deep Learning algorithm that is able to map each world to a proper vector representation.
The approach of mapping a general input, like a word for Word2Vec, in a mathematical object like a vector, is very versatile and it opens different possibilities. For instance, in Word2Vec, it is possible to define a concept of distance between words: "dog" and "cat" will have closer vector representations with respect to "dog" and "pencil". Moreover, since we are dealing with mathematical objects, vector arithmetic can be applied: the vector representation of "queen" can be obtained as the operation "king"-"man"+"woman" or "Italy" as "France"-"Paris"+"Rome". This is just an example of how powerful Deep Learning can be.

2.2 Neural Networks

Among the plethora of algorithms used in Machine Learning, Neural Networks will be widely used in this thesis. For this reason, this section is dedicated in their definition and their application.

2.2.1 The perceptron

In order to define a Neural Network it is essential to firstly define a perceptron, the core architecture of all Neural Networks.
Let's properly define the most simple implementation of a perception conceptualized by Frank Rosenblatt in 1957 [11]. In this implementation, a perceptron is a binary classifier: given an input x, the algorithm will output the label 0 or 1.
Formally, a perceptron is defined by a set of weights w, and a non-linear activation function σ. By denoting as $\hat{y}$ the label produced by the perceptron, it is obtained by:

$$\hat{y} = \sigma\left(\begin{bmatrix}1, x_1, \cdots, x_n\end{bmatrix} \cdot \begin{bmatrix}w_0 \\ w_1 \\ \vdots \\ w_n\end{bmatrix}\right) \tag{2.1}$$

where $\sigma : \mathbb{R} \to \{0,1\}$. A possible implementation for σ could be the step function defined as:

$$\sigma(x) = \begin{cases}0 \;\; if \; 0 > x \\ 1 \;\; if \; x \geq 0\end{cases} \tag{2.2}$$

A graphical representation of how the perceptron works can be seen in Figure 2.1.

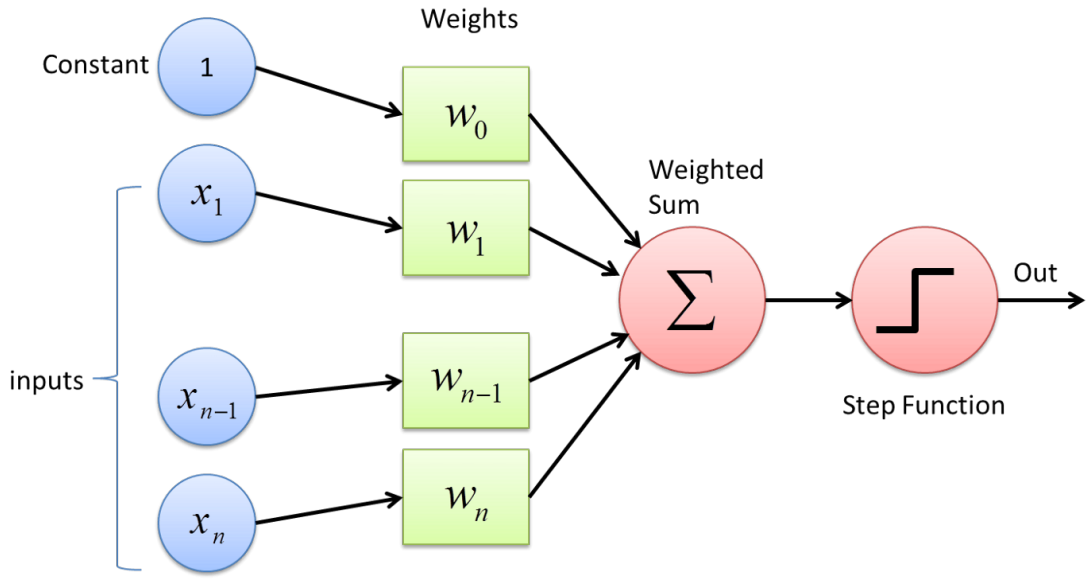

Figure 2.1: Visual representation of a Multi-layer Perceptron

In order to correctly classify the input x with the label 0 or 1 the weights vector w must contain the proper values. In other words, we need to "train" the perceptron in order to get the proper vector w capable of correctly classifying the input.
The standard training process for a perceptron (but for every Neural Network in general) is based on the so called *training set*, a set of data $\{x_1, \cdots, x_n\}$ coupled with the labels $\{y_1, \cdots, y_n\}$. The idea is to iteratively update the vector w in such a way that, the classification error produced by the perceptron will be as little as possible.
The training process for a perceptron can be expressed by the following algorithm: The most important part of Algorithm 1 is on line 6: if the perceptron incorrectly

Algorithm 1 Training of a perceptron

1: $\{x_1, \cdots, x_n\}$,$\{y_1, \cdots, y_n\}$,w,η ▷ Initial data definition
2: **for** $e = 1, \cdots, E$ **do** ▷ Repeat the process for E times
3: **for** $i = 1, \cdots, n$ **do** ▷ Iterate for all data
4: $\hat{y}_i = \sigma(x_i \cdot w^T)$ ▷ Classify x_i
5: **if** $\hat{y}_i \neq y_i$ **then**
6: $w = w + \eta(y_i - \hat{y}_i)$ ▷ If wrong classification, update w
7: **end if**
8: **end for**
9: **end for**

classify the input x_i, it correct itself by updating the weights vector w. The update

is scaled by a factor η, commonly called *learning rate* which defines "how fast" the weight update can happen. Although in a binary classification scenario is not important, the weights are updated by $y_i - \hat{y}_i$ which is the difference between the correct label and the predicted one. In a binary classification problem this difference can only be discrete, but, in a regression problem for instance, where the label is a real number, the bigger the error, the bigger the weights update will be. This concept of "propagating" the classification error through the weights by updating them is the base of what is called *back propagation*[12], the true backbone for training any Machine Learning algorithm.
It is proven that, if data are linearly separable, the perceptron training will converge to a solution for the weights vector w with no classification error.
But how to deal with non-linearly separable data? The solution is concatenating different perceptrons by creating a sequence of perceptron layers, each one with their own weights, in which the output of the previous layer is the input of the next one. This concept of having more perceptrons in one Neural Network is further discuss in the next paragraph.

2.2.2 Multi-layer Perceptron (MLP)

As previously discussed, the concept of MLP was born from the necessity to generalize the perceptron to those cases in which data are not linearly separable. The idea is to concatenate different perceptrons in such a way that the output of the previous one is the input of the next. An MLP architecture is structured in different layers. The first one is defined as input layer since it will receive the raw input data, the layers in the middle of the network are called hidden layers while the last layer is called output layer since its output will be the output of the entire network. Each layer is composed by a different number of neuron. An MLP architecture is also defined as *Fully Connected Network* since each neuron of a layer is connected to each neuron of the next one.

Let's formally define the workflow of a multi-layer perceptron from the input layer through the output one: as notation, the i-th layer will be defined as l_i, having l_0 as the input layer and l_n as the output one. With the same idea, the output of the i-th layer will be y_n, so the input of the layer l_{i+1} will be y_i, the output of the previous one. Each layer has a proper number of neuron d_i which is also referred as layer dimension. The generic layer l_i will receive an input of dimension d_{i-1} and will produce an output of dimension d_i. For this reason, for each layer is defined a matrix weight W_i of dimension $(d_{i-1} \times d_i)$.
Let's define a generic input x of dimension d_0: the input layer of the network will receive it, and will output y_0. Like for the perceptron, an activation function (generally non linear) is involved. Let's define the activation function of each layer as σ_i. Similarly to the perceptron, the operation that the first layer will compute

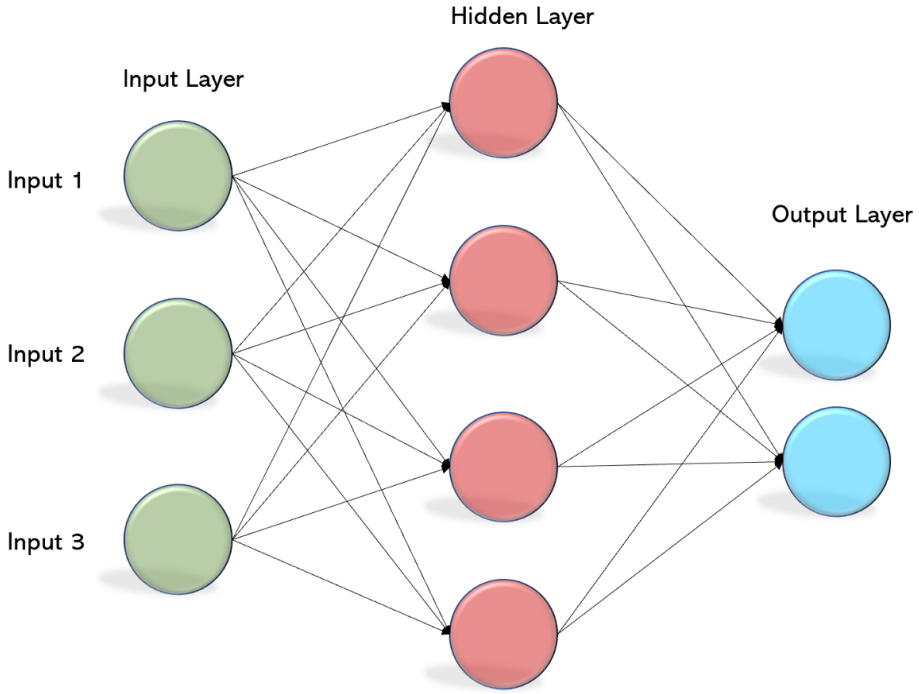

Figure 2.2: Visual representation of a Multi-layer Perceptron

will be:

$$y_0 = \sigma_0(x^T W_0) \tag{2.3}$$

Equation 2.3 can be extended to any layer in the following way:

$$y_i = \sigma_i(y_{i-1}^T W_i) \tag{2.4}$$

Combining all together, the final output of an MLP architecture can be expressed as:

$$y_n = \sigma_n(\sigma_{n-1}(\cdots \sigma_0(x^T W_0)W_1)\cdots)W_n) \tag{2.5}$$

Like for the perceptron, all the weights has to be updated in order to correctly classify the input. The problem is that now the weights are spread among different layers. How can we update each single weight starting from the classification error computed at the end of the Neural Network classification process? An answer to this question is provided in the following section.

2.2.3 How to train a neural network

Just like the perceptron, any neural network has to adjust their weights starting from a classification error. It's possible to generalize this concept by defining a loss

function L which takes in input the classification forecasted by the network $\hat{y}$ and the true label y (or groundtruth) and it produces in output some kind of measure of "how much" error the network as made.
The most important and used loss functions are now reported:

- Regression problems
  1. Mean Square Error (MSE):

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

  2. Mean Square Logaritmic Error (MSLE):

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

  3. Mean Absolute Error (MAE):

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

- Binary classification problems
  1. Binary Cross-Entropy:

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

  2. Hinge Loss:

$$L(y, \hat{y}) = max(0,1 - y\hat{y})$$

  3. Squared Hinge Loss:

$$L(y, \hat{y}) = (max(0,1 - y\hat{y}))^2$$

- Multiclass classification problems
  1. Multiclass Cross-Entropy:

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

2. Hinge Loss:

$$L(y, \hat{y}) = max(0,1 - y\hat{y})$$

3. Kullback Leibler Divergence Loss:

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

By recalling Algorithm 1, the perceptron weights w are updated starting from the error that the perceptron has made $y_i - \hat{y}_i$. Similarly, the update of Neural Network weights must start from a loss function. But how to update different weights in different layers starting from a loss function in order to improve the classification ability of the network? The most common technique used is called *back propagation* and the idea is to back propagate the error computed by the loss function through all the network, from the output layer to the input one.
The back propagation technique is based on the *gradient descent* optimization method. An optimization method is required because the training of a Neural Network can be considered an optimization process where the goal is to minimize the loss function and, to accomplish it, it is necessary the research of optimal weights in the weight space.
The idea of gradient descent is the following: let's "move" the weights in the gradient's opposite direction. By doing so, my weights will be updated towards a minimum of the loss function and the network error will be smaller. Equation 2.6 expresses mathematically this concept.

$$w \leftarrow w - \nabla L(y, \hat{y}) \tag{2.6}$$

Still, gradient descent is only the instrument that back propagation uses to define the direction of the weights update. In order to understand how to compute the actual value of each weight update let's first understand how the loss gradient is computed.
For the sake of simplicity, let's define a simple neural network composed by an input layer, two hidden layers and one output layer, each one with only one neuron.

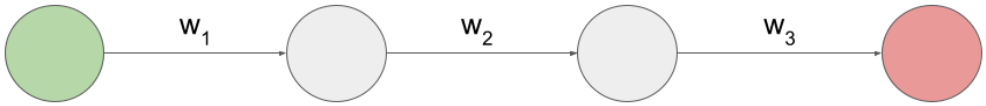

Figure 2.3: Visual representation of a Neural Network with 4 layers, one neuron each

The network depicted in Figure 2.3 is defined by only 3 weights w_1, w_2 and w_3. Let's define the output of each layer L_i as y_i. As previously mentioned, $y_i = \sigma_i(w_i\hat{y}_{i-1})$, or, in other words, each layer will take as input the output of the previous one and then it will multiply it by its weight. To the result, an activation function σ_i is applied. The loss function will be a function of the true value y and the network output $\hat{y}_3$, which can be explicitly expressed as function of the weights as:

$$\hat{y}_3 = \sigma_3(w_3\hat{y}_2) = \sigma_3(w_3\sigma_2(w_2\hat{y}_1)) = \sigma_3(w_3\sigma_2(w_2\sigma_1(w_1x))) \tag{2.7}$$

In the end, each weight has an influence on the network output $\hat{y}_3$, hence, every weight has a contribution on the final loss function $L(y, \hat{y}_3)$. The back propagation technique can be considered a method to quantize this contribution for each weight and adjust it accordingly.
Let's start with the last weight w_3: we would like to understand how much the loss will change if w_3 changes too. Formally we are asking the partial derivative of the loss function by w_3: $\frac{\delta L}{\delta w_3}$. Naturally, this quantity can not be directly computed since the Loss is only a function of $\hat{y}_3$. The workaround for this issue is the *chain rule* mechanism:

$$\frac{\delta L}{\delta w_3} = \frac{\delta L}{\delta \hat{y}_3} \cdot \frac{\delta \hat{y}_3}{\delta w_3} \tag{2.8}$$

Equation 2.8 illustrates that it is possible to compute the partial derivative of the Loss function by a general weight as a chain of product of different partial

derivatives.
Similarly, for the partial derivative of the Loss for w_2 will be:

$$\frac{\delta L}{\delta w_2} = \frac{\delta L}{\delta \hat{y}_3} \cdot \frac{\delta \hat{y}_3}{\delta \hat{y}_2} \cdot \frac{\delta \hat{y}_2}{\delta w_2} \tag{2.9}$$

and, for w_1, without any surprise the partial derivative will be:

$$\frac{\delta L}{\delta w_1} = \frac{\delta L}{\delta \hat{y}_3} \cdot \frac{\delta \hat{y}_3}{\delta \hat{y}_2} \cdot \frac{\delta \hat{y}_2}{\delta \hat{y}_1} \cdot \frac{\delta \hat{y}_1}{\delta w_1} \tag{2.10}$$

In conclusion, an input x is given to the network that will produce $\hat{y}_3$ as result. The error of the network will be computed by the Loss function L which will take, as parameters, the true value y and the network prediction $\hat{y_3}$. In order to update the weights, the gradient descent optimization method has to be applied so: $w \leftarrow w - \eta \nabla L(y, \hat{y}_3)$ (η is the learning rate parameter which regularizes "how much" the weights are updated). In order to compute the gradient $\nabla L(y, \hat{y}_3)$, the chain rule technique of partial derivatives is used, obtaining:

$$\nabla L(y, \hat{y}_3) = \begin{bmatrix} \frac{\delta L}{\delta w_1} \\ \frac{\delta L}{\delta w_2} \\ \frac{\delta L}{\delta w_3} \end{bmatrix} = \begin{bmatrix} \frac{\delta L}{\delta \hat{y}_3} \cdot \frac{\delta \hat{y}_3}{\delta \hat{y}_2} \cdot \frac{\delta \hat{y}_2}{\delta \hat{y}_1} \cdot \frac{\delta \hat{y}_1}{\delta w_1} \\ \frac{\delta L}{\delta \hat{y}_3} \cdot \frac{\delta \hat{y}_3}{\delta \hat{y}_2} \cdot \frac{\delta \hat{y}_2}{\delta w_2} \\ \frac{\delta L}{\delta \hat{y}_3} \cdot \frac{\delta \hat{y}_3}{\delta w_3} \end{bmatrix} \tag{2.11}$$

2.3 Different Neural Network architectures

Neural Networks are a powerful and general purpose instrument. They can be applied in different scenarios, from the image domain to the natural language one. They can solve a wide variety of tasks and output different kinds of outputs: images, text, prediction, classification labels and so on. Of course, there are many kinds of Neural Network architectures intentionally invented to solve particular tasks. In this section, the most important architectures are reported and explained.

2.3.1 Convolutional Neural Network

Convolutional Neural Networks (CNN)[8] are the "de-facto" architecture in the image domain. In computer vision, the concept of convolution was born before the diffusion of Neural Networks. The idea of a CNN architecture is to extend the traditional concept of convolution and integrate it inside a Neural Network.

Before explaining CNNs, it is important to talk about convolution in computer

vision: convolution is generally used to extract or create a feature map out of the input image. This is done with the help of filters (or kernels) that are applied on the input image.
Let's start with a toy example by defining a 5×5 black and white image: to do that, let's write the image with a 5×5 matrix in which each cell is associated to each pixel and its value will span between 0 and 255 (0 is total black and 255 is total white):

$$I = \begin{bmatrix} 0 & 35 & 255 & 255 & 50 \\ 5 & 40 & 255 & 255 & 23 \\ 0 & 15 & 255 & 255 & 10 \\ 12 & 46 & 255 & 255 & 32 \\ 0 & 0 & 255 & 255 & 10 \end{bmatrix}$$

To this image, let's apply the following 3×3 filter:

$$F_1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

How does a filter is applied to an image? The idea is to span the 3×3 filter along all the original matrix by computing a dot product between the filter and each 3×3 sub-matrix of the original image. Image 2.4 shows how a convolutional filter is applied on a filter:

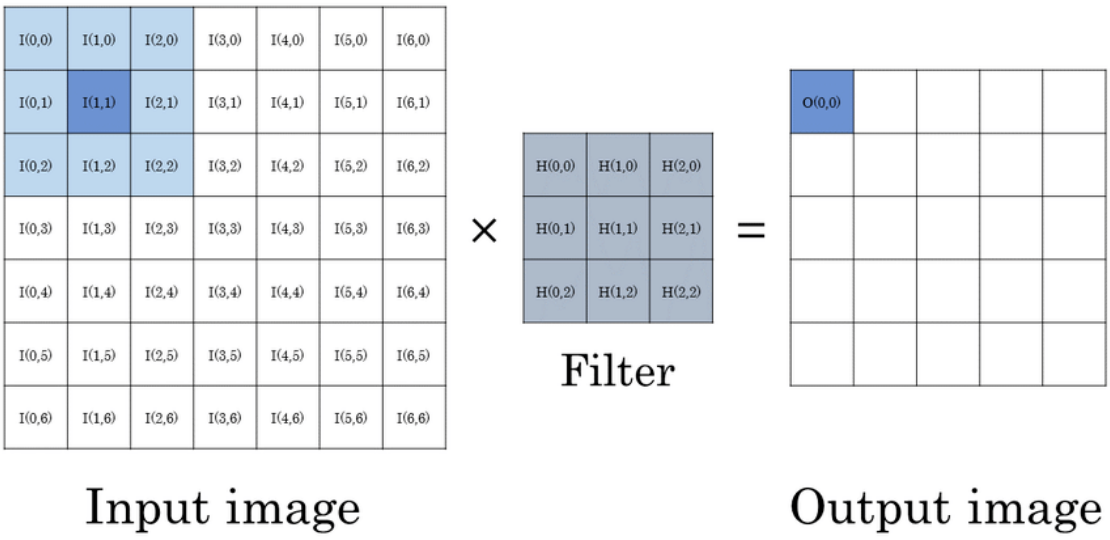

Figure 2.4: How a convolutional filter is applied

If we apply the previous filter on the original image defined before, we obtain a

3×3 image with the following values:

$$O_1 = \begin{bmatrix} 150 & -215 & -232 \\ 281 & -240 & -245 \\ 98 & -209 & -223 \end{bmatrix}$$

The resulting image has values outside the image domain [0,255], hence the final result has to be clipped, namely all the negative values must be clipped to zero and all the values above 255 must be reduced to it:

$$O_1 = \begin{bmatrix} 150 & 0 & 0 \\ 255 & 0 & 0 \\ 98 & 0 & 0 \end{bmatrix}$$

Mathematically, $O_1 = I \bigotimes F_1$ where $\bigotimes$ is the convolution operator that applies a filter on an image. O_1 can be considered a particular feature of the image I under the filter F_1. Naturally, another image will obtain another output by applying the same filter on it. By changing the filter, the feature extracted from the images on which it is applied will change.

A natural question arises: what are the best filters to apply on images in order to retrieve the most important features? Are they depending on the type of the images, or on the task? Before Machine Learning, in computer vision, those filters were handcrafted. For instance, the filter defined before is called *sharpen filter* and it was created with the specific goal of highlighting the edges in an image.

The innovation of the CNN architecture is based on the automatic learning of those filters. The idea is to consider the filter values as weights of the Neural Network and, exactly like every weight, they will update thanks to the back-propagation process. The advantage of this approach is that, for every specific task, the best filters will be automatically applied on the images without relying on some handcrafted kernels.

As stated before, CNN is the standard architecture in the image domain. But how to deal with a task like image classification? Convolution is not enough. The idea is to exploit convolutions in order to extract features from the images and also reduce the dimension of the problem (it is important to note that, in the previous example, the original image was 5×5 while O_1 was only a 3×3) and, starting from those extracted features, classify the image with a sequence of dense layers (like the multi layer perceptron explained before). Recalling the previous example, the matrix O_1 has to be flattened:

$$O_1 = \begin{bmatrix} 150 & 0 & 0 \\ 255 & 0 & 0 \\ 98 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 150 & 255 & 98 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$$

This new vector will be the input of a classifier like a multi-layer perceptron that will result in assigning a label for the image.

2.3.2 Recurrent Neural Network

If CNN is the standard architecture in the image domain, Recurrent Neural Network (RNN)[13] can be considered the standard in the natural language domain. One of the main reason for that is the fact that RNNs are able to manage input of dynamic size. In the natural language domain, in fact, the data to work on are sentences and not all the sentences have the same length.
As depicted in Image 2.5, the RNN workflow can be seen as an iterative process composed in cycles. At each cycle, the output of the RNN will become the input of the next one.

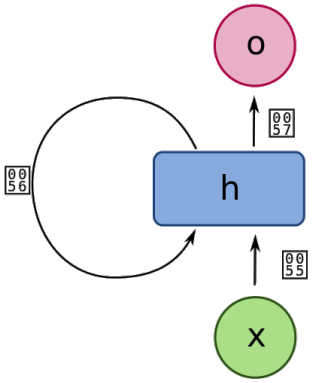

Figure 2.5: Visual representation of an RNN

In Image 2.5, with x is denoted the input, h represents the hidden layers inside the network while with o is defined the output of the network.
This kind of architecture is very versatile in its structure. There are, in fact, a lot of different versions of an RNN architecture reported in Image 2.6.

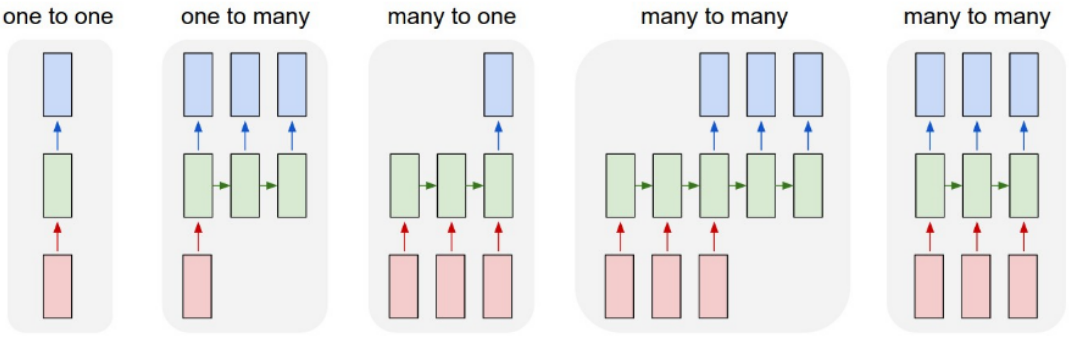

Figure 2.6: Different versions of an RNN

Each one of them can solve specific tasks. To cite some of them, an architecture *many to many* can be used for the *text translation* task in which a sentence of an arbitrary length in the source language is mapped to a sentence of an arbitrary length in the target language.
A *one to many* architecture can be used for text generation: given the first word as input, the next words of the sentence will be computed by the network.
A *many to one* architecture instead, can be used for *sentiment analysis*, where, given a sentence of an arbitrary length as input, the RNN will produce as result the sentiment of that sentence.

2.3.3 Generative Adversarial Network

Generative adversarial networks (shortly GANs) [14], are a particular Neural Network architecture used to generate data. More specifically, given data in input, the goal of a GAN network is to generate other samples similar to the ones received in input. For this reason, the GANs are defined as generative models.
The architecture of a GAN, as reported in Figure 2.7, is composed by two components: the generator and the discriminator.

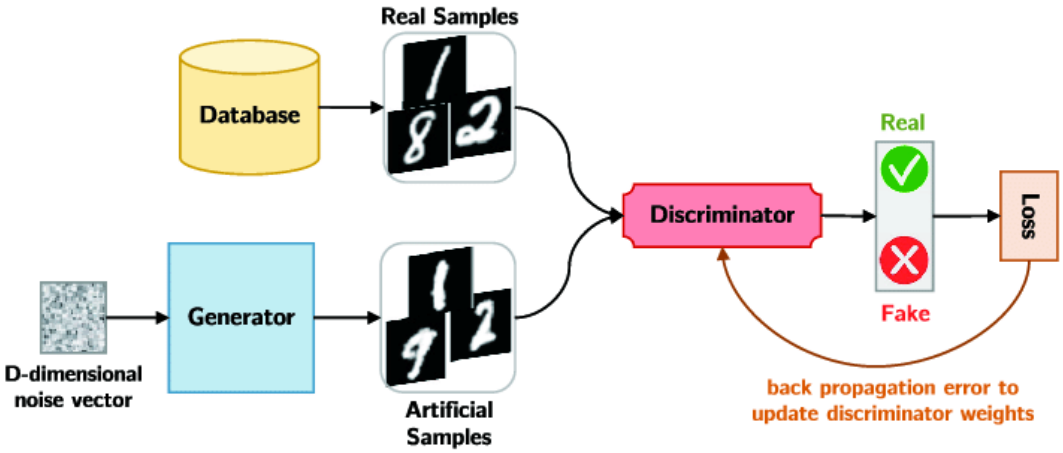

Figure 2.7: Architecture of a GAN

A GAN network can be seen as a two-player non-cooperative game [15]: starting from random noise, the generator produces artificial samples as close as possible to the real ones. The generator then randomly receives real or fake examples and its goal is to distinguish between them. Essentially, the discriminator, is a two-class classifier.
Through the classic back-propagation technique, the discriminator will update its weights in order to become more efficient in distinguish between real and fake

examples, while the generator will become more efficient in producing artificial samples very similar to the real ones.
GAN architectures are relevant in the Federated Learning scenario mainly from a privacy point of view: many attacks in FL are based on reconstructing the clients' private data with a GAN [16]. More details will be provided in the next chapters.

Chapter 3

Federated Learning

3.1 Introduction

In a FL typical scenario, it is assumed that K clients $\{C_1, \cdots, C_K\}$ own their datasets $\{D_1, \cdots, D_K\}$, $D_i \in \mathbb{D}$ where $\mathbb{D}$ denotes the data space and each of them cannot directly access to other clients' data to expand their own data. The central server will be denoted as C_0.
A traditional FL training process is divided into R rounds.
In the first round $r = 1$, the central server C_0 initializes the model $\theta_0^1 \in \Theta$ where Θ is the space of all models. Then, it selects a fraction c of clients (the notation $\tilde{C}_r$ will indicate the subset of clients of cardinality $\lfloor cK \rfloor$ selected at round r).
C_0 sends the newly initialized model θ_0^1 to all the selected clients $\tilde{C}_1$. Each client C_i, then, trains the received model θ_0^1 for E epochs, on the local dataset D_i, resulting in the updated model θ_i^1. More generally, the training process on a client C_i at a generic round r will be denoted as:

$$\theta_r^i = T(\theta^*, D_i, E) \tag{3.1}$$

Where $T : \Theta, \mathbb{D}, \mathbb{N} \to \Theta$ denotes the training process that starts from an initial model θ^*, uses the data D_i and lasts E epochs. T has the following properties:

1. $T(\theta, D,0) = \theta$
2. $\theta_1 = T(\theta_0, D, E) \quad \theta_2 = T(\theta_1, D, E) = T(\theta_0, D,2E) \quad \cdots \quad \theta_n = T(\theta_0, D, nE)$
3. $\theta_A = T(\theta, D_A, E) \quad \theta_B = T(\theta, D_B, E) \to T(\theta_A, D_B, E) \neq T(\theta_B, D_A, E)$

After the training computed by all the selected clients, each one of them will send the trained model to the central server C_0, resulting in a collection of models $\{\theta_i^1 \ \ \forall i \in \tilde{C}_1\}$.

The central server C_0 then aggregates all the incoming models through an aggregation process $A : \{\Theta\} \rightarrow \Theta$, resulting in the creation of a new model that will serve as starting point for the next round:

$$\theta_0^2 = A(\{\theta_i^1 \ \ \forall i \in \tilde{C_1}\}) \tag{3.2}$$

Now the second round starts ($r = 2$), hence, a new subset of clients $\tilde{C_2}$ must be selected and the process repeats.
The goal of the aggregating function A is aggregates models trained on different datasets in order to emulate a model trained on the union of those data:

$$\theta_A = T(\theta, D_A, E) \quad \theta_B = T(\theta, D_B, E) \rightarrow A(\{\theta_A, \theta_B\}) = T(\theta, \cup\{D_A, D_B\}, E) + \epsilon \tag{3.3}$$

Where $\epsilon > 0$ denotes the distance between the model trained on the whole dataset with the aggregated one.
We can define, for comparison, the standard centralized approach applied in ML.

$$\theta_C = T(\theta_0^1, \cup_{i=1}^K D_i, ER) \tag{3.4}$$

where θ_C denotes the trained model in a centralized fashion and θ_0^1 denotes the same randomly initialized model used for the FL training.
The number of epochs is set to ER in order to guarantee the same amount of computation for both centralized and federated scenarios since, in FL, for each round, every client computes E epochs of training. The difference between the federated and the centralized scenarios is the fact that, in FL, for each round only a portion c of clients will be selected (and consequently only a portion of the data will be available for the model), while, in the centralized approach, all the data will be available at each epoch.
As a consequence, equation 3.4 can be fairly compared only with a FL training process in which $c = 1$.

3.1.1 Formal comparison between federated and centralized approach

Assuming $c = 1$, the centralized model θ_C and the federated model θ_0^R have been trained on the same data for the same amount of training computations. What FL aims to achieve is $\theta_0^R = \theta_C$. This is not an easy task due to the fact that, in a FL setup, data are split in K local datasets $D_1, \cdots, D_K$ with independent class distributions. In a centralized scenario we have:

$$T(\theta^{n+1}, \cup_{i=1}^K D_i, E) = T(T(\theta^n, \cup_{i=1}^K D_i, E), \cup_{i=1}^K D_i, E) = \\ T(\theta^n, \cup_{i=1}^K D_i, 2E) = \cdots = T(\theta^0, \cup_{i=1}^K D_i, nE) \tag{3.5}$$

Equation 3.5 is justified by property 2 of T.
In a FL scenario we have:

$$T(\theta^{n+1}, \cup_{i=1}^{K} D_i, E) = T(A(\{\theta_0^n, \cdots, \theta_K^n\}), \cup_{i=1}^{K} D_i, E) = \\ T(T(\theta^n, \cup_{i=1}^{K} D_i, E) + \epsilon^n, \cup_{i=1}^{K} D_i, E) = \cdots = T \cdots T(T(\theta^0, \cup_{i=1}^{K} D_i, E) + \epsilon^0, \cup_{i=1}^{K} D_i, E) \tag{3.6}$$

In order to align the two scenarios, we need $\epsilon^i \to 0 \; \forall i$.
Related to the *heterogeneity* problem it is important to remember the following:

$$||\theta_A - \theta_B||^2 \propto ||P_A - P_B||^2 \tag{3.7}$$

where θ_A and θ_B are two models trained on D_A and D_B respectively, while P_A and P_B are the distributions of the two datasets. Essentially, a model "reflects" the data upon which is trained, hence two models trained on different datasets are different themselves.
The distance ϵ between the centralized model and the aggregated model through the aggregating function A, has a strict relation with the Equation 3.7. In particular:

$$\epsilon \propto ||\theta_A - \theta_B||^2 \propto ||P_A - P_B||^2 \tag{3.8}$$

The more D_A and D_B have a different distribution, the more the aggregating distance to the centralized model ϵ is big, hence, the harder will be to the FL setting to keep up with the centralized one.

3.2 Related works

3.2.1 Federated Learning Taxonomy

In FL, different works aim to improve the standard and first algorithm proposed in the field, namely FedAvg [5]. They can be divided in different categories:

1. **Regularization methods**: Clients with models too different w.r.t. the averaged one are penalized
2. **Clustering methods**: The general idea is to group different clients together according to some rule. Inside this category we can distinguish:
   (a) **Multitask methods**: Clients with similar data distributions are clustered together. Each cluster is considered a separated task and a dedicated model is trained among only those clients.
   (b) **Sequential methods**: Clients with different data distributions are clustered together. A model is then trained for each cluster. Each model then will see data among all the classes.

3.2.2 FedAvg

In this section the FedAvg algorithm introduced in [5] is formally defined.
At each round $0 \leq r \leq R$, the central server C_0 sends the model θ_0^r to the fraction of selected clients $\tilde{C}_r$. Each selected client $C_i \in \tilde{C}_r$ will then train the received model on its own local data by computing $\theta_i^r = T(\theta_0^r, D_i, E)$. All the models $\theta_i^r \ \forall \ i \ s.t. \ C_i \in \tilde{C}_r$ are sent back to the server C_0 that starts the aggregation process:

$$\theta_0^{r+1} = A(\{\theta_i^r \ \forall \ i \ s.t. \ C_i \in \tilde{C}_r\}) = \sum_{i \ s.t. \ C_i \in \tilde{C}_r} \frac{|D_i|}{|D|} \theta_i^r \tag{3.9}$$

Equation 3.9 defines the aggregation function of FedAvg as the weighted sum of the locally trained models weighted by the size of the local datasets ($|D_i|$) normalized ($|D| = \sum_{i=1}^{K} |D_i|$). Algorithm 2 shows in pseudo-code the algorithm.

Algorithm 2 FedAvg Algorithm

```
C_0, {C_1, ··· , C_K}, R, θ_0^0                ▷ Initial data definition
for r = 1, ··· , R do
    C̃_r = random({C_1, ··· , C_K}, ⌊cK⌋)       ▷ Select a random subset of clients
    for C_i, ∈ C̃_r do                           ▷ In parallel, on client side
        θ_i^r = T(θ_0^r, D_i, E)
    end for
    θ_0^r = Σ (|D_i|/|D|) θ_i^r                 ▷ Aggregation on server side
end for
return θ_0^R                                    ▷ θ_0^R is the trained model
```

3.2.3 Regularization methods

In this section the regularization methods in FL will be analyzed. The general idea in common among all the methods is correcting the 'shift' that some clients may produce due to their data distribution that could lead to poor global performances. Among all the algorithms the most important ones are FedProx [6], FedDyn[17] and SCAFFOLD[18].

FedProx

FedProx changes the loss function that each client minimizes during the training phase by adding a regularization term that constrain the model to be 'close' to the model received from the server at the beginning of the round. Formally:

$$\theta_i^{r+1} = min_\theta \ L_i(\theta_0^r) + \frac{\mu}{2} ||\theta - \theta_0^r||^2 \tag{3.10}$$

where $L_i(\theta_0^r)$ is the loss function of the client C_i that has to be minimized in the training phase resulting in the updated model θ_i^{r+1}. $||\theta - \theta_0^r||^2$ is the regularization term that force the model θ to be 'close' to θ_0^r. This term is weighted by an hyperparameter μ. Algorithm 3 shows in pseudo-code the algorithm.

Algorithm 3 FedProx Algorithm

C_0, $\{C_1, \cdots, C_K\}$, R, θ_0^0 ▷ Initial data definition
for $r = 1, \cdots, R$ **do**
  $\tilde{C}_r = random(\{C_1, \cdots, C_K\}, \lfloor cK \rfloor)$ ▷ Select a random subset of clients
  for $C_i, \in \tilde{C}_r$ **do** ▷ In parallel, on client side
    $\theta_i^r = T(\theta_0^r, D_i, E) \rightarrow min_\theta \; L_i(\theta) + \frac{\mu}{2}||\theta - \theta_0^r||^2$
  end for
  $\theta_0^r = \sum \frac{|D_i|}{|D|} \theta_i^r$ ▷ Aggregation on server side
end for
return θ_0^R ▷ θ_0^R is the trained model

FedDyn

Also FedDyn exploits the same idea of FedProx in order to control the client shift phenomenon. In particular FedDyn add to the local loss function two regularization terms (linear and quadratic):

$$\theta_i^{r+1} = min_\theta \; L_i(\theta_0^r) - \langle \nabla L_i(\theta_i^r), \theta \rangle + \frac{\alpha}{2}||\theta - \theta_0^r||^2 \tag{3.11}$$

where $\langle \nabla L_i(\theta_i^r), \theta \rangle$ is the linear regularization term and, equally to FedProx, $\frac{\alpha}{2}||\theta - \theta_0^r||^2$ is the quadratic one, weighted by α. At each iteration, the loss gradient $\nabla L_i(\theta_i^r)$ is updated as:

$$\nabla L_i(\theta_i^{r+1}) = \nabla L_i(\theta_i^r) - \alpha(\theta_i^{r+1} - \theta_0^r) \tag{3.12}$$

FedDyn changes also on server side w.r.t FedAvg by computing the aggregation function as:

$$\theta_0^{r+1} = A(\{\theta_i^r \; \forall \; i \; s.t. \; C_i \in \tilde{C}_r\}) = \left(\frac{1}{|\tilde{C}_i|} \sum_{i \; s.t. \; C_i \in \tilde{C}_r} \theta_i^{r+1} \right) - \frac{1}{\alpha} h^{r+1} \tag{3.13}$$

where $h^{r+1} = h^r - \alpha \frac{1}{|\tilde{C}_i|} \left(\sum_{i \; s.t. \; C_i \in \tilde{C}_r} \theta_i^{r+1} - \theta_0^r \right)$.
FedDyn is depicted in Algorithm 4.

Algorithm 4 FedDyn Algorithm

C_0, $\{C_1, \cdots, C_K\}$, R, θ_0^0 ▷ Initial data definition
for $r = 1, \cdots, R$ **do**
  $\tilde{C}_r = random(\{C_1, \cdots, C_K\}, \lfloor cK \rfloor)$ ▷ Select a random subset of clients
  for $C_i, \in \tilde{C}_r$ **do** ▷ In parallel, on client side
    $\theta_i^r = T(\theta_0^r, D_i, E) \rightarrow min_\theta \; L_i(\theta) - \langle \nabla L_i(\theta_i^{r-1}), \theta \rangle + \frac{\alpha}{2}||\theta - \theta_0^{r-1}||^2$
    $\nabla L_i(\theta_i^r) = \nabla L_i(\theta_i^{r-1}) - \alpha(\theta_i^r - \theta_0^{r-1})$
  end for
  $h^r = h^{r-1} - \alpha \frac{1}{|\tilde{C}_i|} \left(\sum_{i \; s.t. \; C_i \in \tilde{C}_r} \theta_i^r - \theta_0^{r-1} \right)$
  $\theta_0^r = \left(\frac{1}{|\tilde{C}_i|} \sum_{i \; s.t. \; C_i \in \tilde{C}_r} \theta_i^r \right) - \frac{1}{\alpha} h^r$ ▷ Aggregation on server side
end for
return θ_0^R ▷ θ_0^R is the trained model

SCAFFOLD

SCAFFOLD introduces the concept of "client control variate" as a method to steer, at each round, all the clients towards solutions not too far from the initial server model. Essentially, each client C_i is coupled with a client control variate c_i. Also the server is coupled with a server control variate c. SCAFFOLD modifies the standard Stochastic Gradient Descent used during the training phase by adding the term $c_i - c$ in order to constrain the client model θ_i^r more towards the server model θ_0^r. Revisiting Equation [SGD Eq.], SCAFFOLD redefines the SGD method as:

$$\theta_i^{e+1} = \theta_i^e + \nu(\nabla \theta_i^e + c - c_i) \tag{3.14}$$

The client control variate c_i can then be updated by following two possible strategies:

$$\begin{cases} (1) & c_i^+ = \nabla(\theta_0^r) \\ (2) & c_i^+ = c_i - c + \frac{1}{E\nu}(\theta_0^r - \theta_i^r) \end{cases} \tag{3.15}$$

Option (1) involves making an additional pass over the local data to compute the gradient at the server model θ_0^r. Option (2) instead re-uses the previously computed gradients to update the control variate. Option (1) can be more stable than (2) depending on the application, but (2) is cheaper to compute and usually suffices. On server side, SCAFFOLD does the following:

$$\begin{cases} (1) & \theta_0^{r+1} = \frac{\nu}{|\tilde{C}_r|} \sum_{i \; s.t. C_i \in \tilde{C}_r} \theta_i^r - \theta_0^r \\ (2) & c = c + \frac{1}{N} \sum_{i \; s.t. C_i \in \tilde{C}_r} c_i^+ - c_i \end{cases} \tag{3.16}$$

Firstly, with (1) it defines the aggregating function used in order to create the new server model for the next round $r + 1$. Secondly it updates the "server control variate" c by equation (2).

Algorithm 5 SCAFFOLD Algorithm

C_0, $\{C_1, \cdots, C_K\}$, R, θ_0^0, c ▷ Initial data definition
for $r = 1, \cdots, R$ **do**
  $\tilde{C}_r = random(\{C_1, \cdots, C_K\}, \lfloor cK \rfloor)$ ▷ Select a random subset of clients
  for $C_i, \in \tilde{C}_r$ **do** ▷ In parallel, on client side
    for $e = 1, \cdots, E-1$ **do** ▷ Explicit definition of the training process
      $\theta_i^{e+1} = \theta_i^e + \nu(\nabla\theta_i^e + c - c_i)$
    end for
    $\theta_i^r = \theta_i^E$
    (1) $c_i^+ = \nabla(\theta_0^r)$ or (2) $c_i^+ = c_i - \frac{1}{E\nu}(\theta_0^r - \theta_i^r)$
  end for
  $\theta_0^{r+1} = \frac{\nu}{|\tilde{C}_r|} \sum_{i\ s.t. C_i \in \tilde{C}_r}$ ▷ Aggregation on server side
  $c = c + \frac{1}{N} \sum_{i\ s.t. C_i \in \tilde{C}_r} c_i^+ - c_i$
end for
return θ_0^R ▷ θ_0^R is the trained model

3.2.4 Clustering methods

As the name said, clustering methods group clients together according to some rule [19]. Generally, the rule is based on the clients' data distribution. Following the FL paradigm, due to privacy constraints, it is forbidden to directly access clients' data distribution. For this reason the data distribution of each client has to be indirectly inferred through some metric. Once all the clients distributions are approximated without directly accessing the clients' data, the clusters can be composed according with some predefined rule.
Formally, let's define a clustering rule G as an ensamble of three elements:

1. *Client distribution approximator* $\psi(.)$: it provides statistics regarding the local distribution of a client in a privacy-preserving way

2. *Metric* τ : it evaluates the distances between the estimated data distributions

3. *Grouping method* $\phi(.)$: it defines the rule through which the clients will be clustered together

So $G := \{\psi, \tau, \phi\}$. The standard procedure through which clients are clustered together follows the following steps:

1. The server C_0 applies the approximator ψ to all clients $(\{\psi(C_1), \cdots, \psi(C_K)\})$ and obtains the approximations of the local data distributions $(\{\tilde{P}_i \approx P_i\}_{i=1}^K)$.

2. Through the metric τ, the distances between all the approximations $\tilde{P}_i$ are computed, obtaining a distance matrix $D_{K\times K}$ where the $i-j$ element is $\tau(\tilde{P}_i, \tilde{P}_j)$.

3. From D, the grouping method ϕ is applied, resulting in the realization of N clusters of clients $\{S_1, \cdots, S_N\}$.

Algorithm 6 depict the aforementioned clustering method general idea:

Algorithm 6 Clustering methods

$\{C_1, \cdots, C_K\}$, ψ, ϕ, τ ▷ Initial data definition
$\{\tilde{P}_1, \cdots, \tilde{P}_K\} \leftarrow \{\psi(C_1), \cdots, \psi(C_K)\}$ ▷ Server sends the approximator to all the clients
$D = [0]_{K\times K}$ ▷ Distance matrix initialization
for $i = 1, \cdots, K$ **do**
    for $j = 1, \cdots, K$ **do**
        $D_{i,j} = \tau(\tilde{P}_i, \tilde{P}_j)$
    end for
end for
$\{S_1, \cdots, S_N\} \leftarrow \{C_1, \cdots, C_K\}$ ▷ Creation of N clusters

Multitask methods

Among the clustering methods, Federated Multitask learning (FMTL) methods are the ones that follow the following principle: cluster similar clients together and train one specialized model per per group. This approach relies on the underlying assumption that each cluster will represent a different task, hence it is convenient to have a specialized model tuned for that particular task.

Inside this category we can find algorithms like **CFL** [20] where clients are iteratively bi-partitioned in groups with the goal of putting similar clients together. The metric τ used in CFL is the cosine similarity, while, as approximation of the local data, this algorithm trains, for each client an initialized model θ and then it uses its gradient as approximation.

Another example of FMTL algorithm is **FeSEM** [19]. This work has some similarities with the notorious clustering algorithm K-means[21] applied in the models weight space. Essentially, at the beginning of the algorithm are randomly defined N "centroid" models $\tilde{\theta_1}, \cdots, \tilde{\theta_N}$. Then, the training process begins with a dual optimization: all the clients model θ_i^r are trained by constraining the model to be close to the nearest centroid but far to all the others. By iterating with this approach, clusters of similar models naturally emerges.

Sequential methods

By relying on clients clustering, this typology of methods goes in the opposite direction of FMTL. The idea is to cluster together clients with different data distributions in order to obtain groups of clients that, internally, have a more homogeneous data distributions obtained as the combination of the internal data distributions of the clients inside the same cluster.
Let's develop this concept with a toy example: let's suppose to have 4 clients c_1, c_2, c_3 and c_4. Let's also suppose to deal with a 2-class problem, hence the clients data distribution vectors $p_1, \cdots, p_4$ will have only two entries. Let's assume that those vectors are defined as $p_1 = [0.1,0.9]$, $p_2 = [0.2,0.8]$, $p_3 = [0.9,0.1]$ and $p_4 = [0.7, 0.3]$. It is obvious in this case that c_1 and c_2 are more similar, because both of them have more data on the second class and fewer data on the first one, while c_3 and c_4 are more similar because of a data preponderance on the first class. The idea in the sequential methods is to group together dissimilar clients, so in this case, a possible clusterization could lead to $S_1 = \{c_1, c_3\},\ S_2 = \{c_2, c_4\}$.
For the sake of simplicity, let's assume that each client's local dataset has a cardinality of n. So the cardinality of S_1's local dataset will be $2n$, divided in $p_{S_1} = \frac{1}{2}(p_1 + p_3) = [0.5,0.5]$. For S_2 we will get $p_{S_2} = [0,45,0,55]$.
The idea of the sequential methods is to train a model per cluster, specifically a model for S_1 and a model for S_2. This approach leads to different advantages: the fact that the model will be trained inside an homogeneous cluster means that the model will see a fair amount of data for each class without being too specialized in some classes rather than others. The second advantage relies on the fact that, if I train a model inside a single client, the amount of data that the model is capable to see is n, while a model trained on the entire cluster will double the data for the training resulting in a more performing model.
FedSeq[7] belongs in this category. The next part of the thesis will be dedicated to this algorithm.

3.3 Privacy in Federated Learning

The main reason why the Federated Learning paradigm has been created is related to privacy: clients private data must be inaccessible from the server or any malevolent attacker. For this reason, a lot of effort has been made in order to guarantee privacy in FL, by developing different defence strategies and by studying different typologies of attacks that can compromise the security of private data.
In the next part the most important attacks and the most important defences will be presented.

3.3.1 Privacy attacks in Federated Learning

If we talk about privacy attacks in FL, we first need to understand who can make such attack [22]. The attacker can be an *insider*, like a malevolent client or the server itself, or it can be an *outsider*, like model consumers or eavesdroppers that can only have access to the model's data. Surely, the attacks from the former is much more effective since an insider has the chance to get the model during the training rather than having only the final model.

Another distinction to make while talking about privacy attacks, is what is attacked. We can split all the attacks in *passive* or *active* attacks. The former are all that attacks that do not interfere with the training of the FL model, because their only goal is to sniff private information. The latter, instead, are all that attacks that do interfere with the FL model training, aiming at lowering the final performances.

It is also important to understand when the attack is made: it can be made during the *training phase* (generally these attacks are stronger) or during *inference phase* by exploiting the final model.

Another important factor that can distinguish an attack from another is where to attack. Essentially, an attack can exploit three elements: the weight update of the model, the gradient update of the model, or directly the trained model. The first two elements lead to more effective attacks while the third one is used only if the attacker is an outsider.

The last important element to define a privacy attack is the "why": what my attack is aiming to do? In this case it is difficult to make an exhaustive list because the reason depends on the task, but it is possible to divide the attacks in this four categories:

1. The inference of class representatives aims to generate representative samples, which are not real data instances of training datasets but can be used to study sensitive information about the training datasets

2. The inference of memberships aims to determine whether a data sample has been used for model training

3. The inference of properties of the training data aims to infer the property information regarding the training datasets

4. The inference of inferring training samples and labels aims to reconstruct the original training data samples and the corresponding labels

3.3.2 Privacy defences in Federated Learning

The Federated Learning research community has invested a lot of effort in creating good countermeasures to the aforementioned attacks [22]. These defences can be divided in four categories:

1. Encryption based defences
2. Perturbation based defences
3. Anonymization based defences
4. Hybrid methods

Encryption based defences

In this category belong all those defences that rely on encryption cryptographic techniques for privacy preservation. Among them we have the *Homomorphic encryption* [23], a technique that guarantees that the same calculations computed on the encrypted data and on the plain data lead to the same result. This techniques is used, for instance, in [24] to encrypt the gradient update of the model.
Another encryption techniques is called *secret sharing*[25] and it guarantees that, given a "secret" to encrypt divided in n shards, this "secret" can be reconstructed only with, at least, $m <= n$ shards. This technique is the FL paradigm in [26].
Another encryption technique widely used to secure privacy in FL is called *Secure multiparty computation (SMC)*. Essentially, SMC is a cryptographic scheme that enables distributed participants to collaboratively calculate an objective function without revealing their own data. This technique has been used in different works like [27].
Generally speaking, this category of defences do not degrade the quality of the resulting model after the FL training process, but it has a significant cost in terms of complexity due to the encryption operations required. This could be a problem since, in FL, we are in a situation of heterogeneous clients in term of computation power, so, some strugglers may not be able to bare the encryption process required by these techniques.

Perturbation based defences

In this category, all the defences are based on the principle of perturbing, in some way, the data that an attacker needs in order to sniff information, without downgrading too much the model performances. One way to accomplish this is through the *global differential privacy techniques*[28]. Specifically, during each training round, the server selects a random number of participants to train the global model, and the participants update their local models and send weights

back to the server. The server then aggregates the global model by adding random Gaussian noise. In this way, malicious participants cannot infer the information of other participants from the shared global model.
An alternative technique is called *local differential privacy*[29]: very similar to the global one, a random Gaussian noise is added to the model gradient, but in this case this operation is computed on client side. This means that the server will never receive the original model.
Belonging to the perturbation based defences there are two more techniques: *Additive perturbation*[30] and *Multiplicative perturbation*[31]. Both techniques aims in transforming the original data space by shifting (the former) or rotating (the latter) in order to secure sensitive data.
All the defences in this categories relies on data perturbation. Although they solve the computation overhead problem posed by the encryption based defences, they present a quality degradation issue.

Anonymization based defences

Anonymization techniques are mainly used to achieve group-based anonymization by removing the identifiable information while maintaining the utility of the published data. There are three types of widely used anonymization techniques: *k-anonymity*[32], *l-diversity*[33], and *t-closeness*[34]. These methods are based on the assumption that data are divided in three categories: unique identifiers (UIDs), sensitive attributes (SAs), and non-sensitive attributes. The defences in this category aim at protecting unique identifiers and sensitive attributes while maintaining the utility of the published data.

Chapter 4

FedSeq

This chapter will be dedicated to a deep analysis of the algorithm FedSeq[7], the core part of this thesis. After a formal and exhaustive definition of the algorithm, several experiments will be presented in order to empirically understand the algorithm performances, compared to the other state of the art (SOTA) methods.

4.1 The algorithm

If we take the Federated Learning algorithms taxonomy previously discussed, FedSeq belongs in the clustering sequential methods category. Indeed, this algorithm clusters clients together with the goal of building groups of clients that, overall, has an homogeneous data distribution. Then, on this group of clients, from now on called *superclients*, the classical FedAvg algorithm is applied. This approach to the Federated Learning paradigm presents two main challenges:

- How to create the superclients without directly access the clients' local data distributions?
- How to train a model inside each superclient?

Let's start with the first question. In order to build superclients in a proper way without violating the FL privacy constraints, FedSeq uses a pipeline of different components:

1. *Client approximator*: denoted with the letter $\psi(.)$, it is used to approximate a client's data distribution with a vector. This step has to be done in a privacy preserving way, so, the standard approach is to leverage on a local model trained on the client's data.
2. *Grouping method*: denoted with the letter $\phi(.)$, it is the component used for cluster the clients together.

3. *Grouping metric*: denoted with the letter τ, it is the metric used to evaluate the distances between the clients' local data approximations.

A schema of this pipeline is reported in Image 4.1.

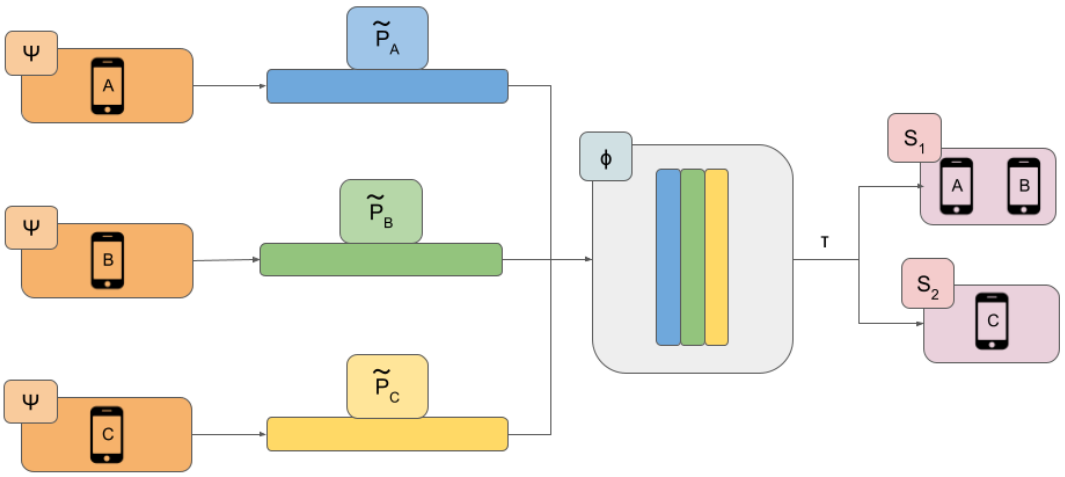

Figure 4.1: FedSeq pipeline for creating superclients

In the next part of this chapter, each one of these elements is discussed.

4.1.1 Client approximator

The goal of the client approximator is to approximate, in a privacy preserving way, the clients' local data distributions. Obviously no local information can be sent to the server from the clients. For this reason the approach used in FedSeq is to exploit a Neural Network and training it on the clients' local data.
More in details, the idea behind exploiting a trained model to approximate data distributions relies on the fact that the trained model itself can be seen as a representation of the data upon which is trained. Given a dataset D, a model $\theta_D = T(\theta, D)$ inherits some properties of D. For instance, in a classification problem, if D has only instances of one class, the model θ_D will classify every possible input as member of that class. In a way, the information that the original data have a very heterogeneous distribution towards one particular class is reflected in the classification ability of the model. That's the principle upon which the client approximator is built.
The client approximator pipeline works in the following way:

1. The server sends to all K clients a randomly initialized model θ_0
2. Each client k trains, for E_0 epochs, the model on its own local data D_k resulting in $\theta_0^k = T(\theta_0, D_k, E_0)$.
3. The server receives all the models $\{\theta_0^k\}_{k=1}^K$ and applies the client approximator ψ on them in order to retrieve the clients' local data approximations $\{\tilde{P}_k\}_{k=1}^K$.

Formally, FedSeq implements two different versions of client approximators:

1. Confidence approximator
2. Weight approximator

Confidence approximator

This approximator relies on a server-side public homogeneous dataset $D_{pub} = \bigcup_{c=1}^{N_c} D_c$ where D_c contains J samples for class c and N_c is the total number of classes. On server side, each model θ_0^k produces a probability vector $p_{k,i,c}$ $k \in [K]$ $i \in [J]$, $c \in [N_c]$, for each example in D_{pub}. Then, for each class c, the following value is computed $p_{k,c} = \frac{1}{J} \sum_{i=1}^{J} p_{k,i,c}$. The final client approximator will result in:

$$p_k = softmax(\{p_{k,1}, p_{k,2}, \cdots, p_{k,N_c}\}) \tag{4.1}$$

Weight approximator

This approximator doesn't require a public dataset D_{pub} to work. It directly exploits the models weights as approximators. Different works [35] have already proven that the model weights can represent the data on which the model was trained on. Naturally, the cardinality of the weights vector of a model can be huge and can raise computational problem during the clustering process. For this reason, in FedSeq, each weights vector w_k is reduced by the P.C.A method [36] conserving 90% of the variance, resulting in:

$$p_k = P.C.A(w_k) \tag{4.2}$$

4.1.2 Grouping method

Once all the clients are approximated by the chosen client approximator, the grouping method $\phi(.)$ comes into play. Its goal is to create different clusters of clients by trying to obtain, for each cluster, an overall data distribution (computed as the sum of the clients' distributions in that cluster) as homogeneous as possible. Image 4.2 shows how the grouping method component works.

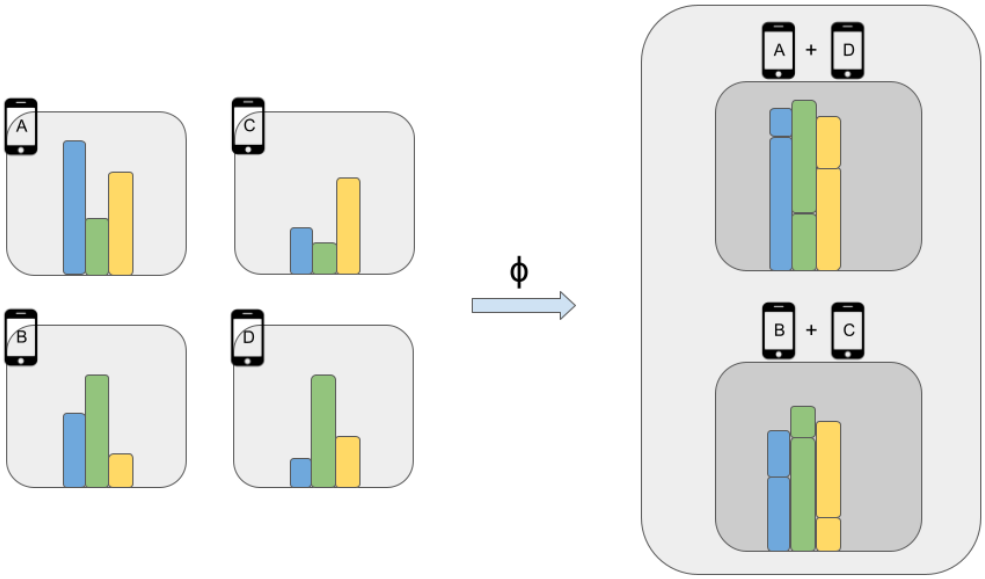

Figure 4.2: How the grouping method component works

Clients A and D are grouped together since the overall amount of data per class is even after the two distributions are merged. With the same logic, also clients B and C are grouped together.
FedSeq implements three different grouping methods: i)ϕ_{rand}, ii)ϕ_{greedy}, iii)ϕ_{kmeans}.

Random Grouping Method

The random grouping method, ϕ_{rand}, is the most naive and basic approach to implement a grouping method. Simply, each cluster is composed by a random selection of clients. The only two constraints that this grouping method has to follow are: i) guaranteeing that each cluster has a minimum number of examples ii) each cluster has a maximum number of clients. These two constraints pose a lower bound and an upper bound to the clusters' dimension. Algorithm 7 codifies this grouping method.

Algorithm 7 Random grouping method

1: $\{C_1, \cdots, C_K\}$, $S = [\,]$, D_{min}, K_{max}, $j = 0$ ▷ Initial data definition
2: **for** $i = 1, \cdots, K$ **do**
3: $S_j = [\,]$
4: $c = random(\{C_1, \cdots, C_K\})$ ▷ Randomly select a client
5: $\{C_1, \cdots, C_K\} \rightarrow c$ ▷ Remove c from the available clients
6: **if** $|S_j| + 1 > K_{max}$ && $|D_{S_j}| + |D_c| > D_{min}$ **then** ▷ If the superclient has too many clients but sufficient examples
7: $S \leftarrow S_j$
8: $j = j + 1$
9: **end if**
10: $S_j \leftarrow c$ ▷ Insert c in the superclient
11: **end for**
12: **return** S ▷ Return the list of superclients

It is important to remember that this grouping method does not guarantee at all the homogeneity of the superclients distributions, but it is used as baseline for the other methods.

Greedy Grouping Method

This grouping method, shortly defined as ϕ_{greedy}, follows, as the name said, a greedy approach: one superclient at a time is created in the best way possible with the available client that it has. More in details: one client is randomly select as starting point. Then, the second client is chosen by finding that client that, together with the first one, result in the most homogeneous overall distribution possible. Then the third client is selected following the same principle: finding that client that maximize the overall distribution homogeneity. Once the superclient respects the two constraints defined before (minimum number of examples and max number of clients), it is ready and the second superclient starts to selects the best clients.
Why *greedy*? Because each superclient selects the best clients in order to maximize its own overall homogeneity distribution without considering the fact that, maybe, a client selected now, could be the optimal choice for another superclient. In conclusion, this approach creates a sub-optimal partitions of clients but it is very feasible from a computational point of view.
Coupled with this grouping method, the metrics through which we evaluate how an available client will affect the homogeneity of the overall superclient's distribution is a key factor of ϕ_{greedy}. In FedSeq, three metric are used: i)*Gini Index* ii)*Kullback-Leibler divergence* iii)*Cosine Distance*. All these metrics will take, as input, the data approximation of a candidate client c and the data approximation of the superclient that we are building (computed as the average of the approximations

of the clients already selected for this superclient).
So each metric will be in the form of: $\tau(D_c, D_S) \to \mathbb{R}$. For the sake of readability, let's define the average between D_c and D_S as $D_{c,s}$.
Regarding the *Gini Index*, τ is defined as:

$$\tau(D_c, D_S) = 1 - \sum_{i=1}^{N} (D_{c,S})_i^2 \tag{4.3}$$

For the *Kullback-Leibler divergence*, instead, τ is defined as:

$$\tau(D_c, D_S) = 1 - \sum_{i=1}^{N} (D_{c,S})_i \cdot \log\left(\frac{(D_{c,S})_i}{1_i}\right) \tag{4.4}$$

The *Cosine Distance* is defined as:

$$\tau(D_c, D_S) = 1 - \frac{D_c \cdot D_S}{|D_c| \cdot |D_S|} \tag{4.5}$$

Algorithm 8 shows the pseudo-code for ϕ_{greedy}.

Algorithm 8 Greedy grouping method

$\{C_1, \cdots, C_K\}$, $S = [\,]$, D_{min}, K_{max}, $j = 0$, τ ▷ Initial data definition
while $|\{C_1, \cdots, C_K\}| > 0$ **do** ▷ Iterate until all clients are clustered
    $S_j = [\,]$
    if $|S_j| == 0$ **then** ▷ First client is randomly chosen
        $c = random(\{C_1, \cdots, C_K\})$
    else ▷ Chose the best client for τ
        $c = argmin(\tau(D_k, D_{S_j}))\ \forall k \in [K]$
    end if
    $S_j \leftarrow c$
    if $|S_j| > K_{max}$ && $|D_{S_j}| > D_{min}$ **then**
        $S \leftarrow S_j$
        $j = j + 1$
    end if
end while
return S ▷ Return the list of superclients

K-means grouping method

The last grouping method implemented in FedSeq is ϕ_{kmeans}. This algorithm can be considered a two-steps process: first, the *k-means* clustering algorithm[21] is applied on the clients' data approximations in order to get clusters of similar clients.

Then, the second step consists of creating the superclients by picking one client per cluster guaranteeing that clients with different data distributions are grouped in the same superclients. Algorithm 9 formalizes this grouping method.

Algorithm 9 K-means grouping method

$\{C_1, \cdots, C_K\}$, $S = [\,]$, D_{min}, K_{max}, $j = 0$, $i = 1$, $h = 1$
$\{G_1, \cdots G_H\} = kmeans(\{C_1, \cdots, C_K\})$ ▷ Creates H clusters with kmeans
while $i <= K$ **do**
    $S_j = [\,]$
    $S_j \leftarrow random(G_h)$ ▷ Random select a client in G_h
    $h = mod(h + 1, H)$ ▷ Get next cluster
    if $|S_j| > K_{max}$ && $|D_{S_j}| > D_{min}$ **then**
        $S \leftarrow S_j$
        $j = j + 1$
    end if
    $i = i + 1$ ▷ Count the number of clients
end while
return S

In this grouping method, the selection of the metric τ is not involved since the *k-means* algorithm can be only defined if the metric used to compute the distances is the euclidean distance.

Chapter 5

Experiments

In this chapter, all the experiments conducted on FedSeq are presented, along with the presentation on the dataset used and the metrics involved in order to evaluate the algorithm. All the experiments can be divided in three categories: i) experiments on the FedSeq parameters that aim to understand what are the best configurations of the algorithm under different conditions ii) experiments that test the algorithm from a privacy point of view that aim to understand how much FedSeq is resilient under a privacy attack and iii) experiments that evaluate the FedSeq performances against other state of the art (S.O.T.A) federated learning algorithms.

5.1 Datasets

In this section, the datasets used in all the experiments are presented. For each of them, the task, a brief description of the classes, and some statistics will be reported.
In order to make experiments in a federated scenario, each dataset has to be split in K local datasets where K is the number of simulated clients. Obviously, it is important to understand how heterogeneous are the K partitions of the datasets. For this reason, for each dataset, it will be explained the partition strategy.

5.1.1 Cifar-10

The **Cifar-10** dataset [37], is a well established dataset in image recognition. It consists of 60.000 images (50.000 in the train set and 10.000 in the test set) coupled with a label that belongs to the 10 classes presented in the dataset. Each image has a dimension of 32×32 and three color channels.
The 10 classes, along with some image examples are reported in Figure 5.1.

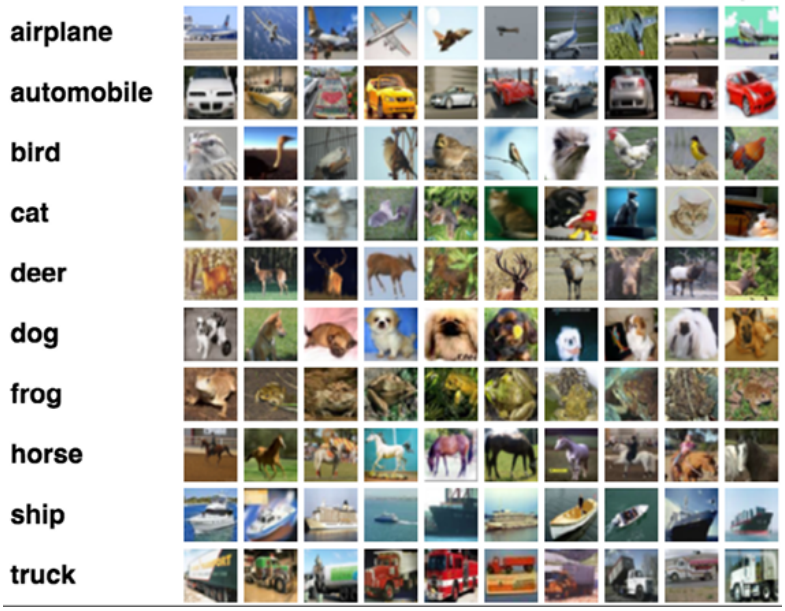

Figure 5.1: The Cifar-10 dataset

The dataset is uniformly distributed, meaning that, for each class, there are 6.000 examples, 5.000 in the training set and 1.000 in the test set.
The partition in K private local datasets is managed with a Dirichlet distribution [38] parametrized by α:

$$p(x) \simeq \prod_{i=1}^{K} x_i^{\alpha} \tag{5.1}$$

The Dirichlet distribution guarantees that, the sum of the K entries sum up to 1. The smaller α, the fewer elements in the distributions will have support. The two extreme cases are $\alpha = 0$, which means that each distribution will have only support in one element and $\alpha \to \infty$ which transforms the Dirichlet distribution to a Uniform distribution. All the experiments conducted on Cifar-10 set a specific α in order to simulate how much heterogeneous clients are.

5.1.2 Cifar-100

The **Cifar-100** dataset [37] can be considered the extension of Cifar-10. The domain and the number of examples are the same: image classification and 60.000 images split in 50.000 for the training set and 10.000 for the test set. The number of classes is 100. They are grouped in 20 superclasses, each one of them containing 5 fine labels. As in Cifar10, the images are uniformly distributed among all the 100 classes, resulting in 600 examples per class (500 in the train set and 100 in the

test set).
Image 5.2 reports the classes with an example.

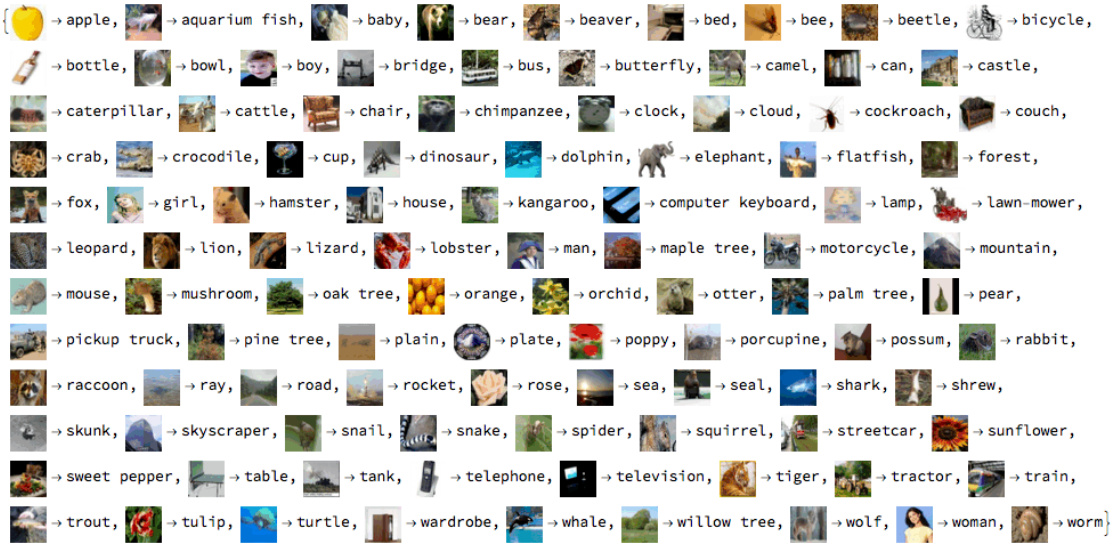

Figure 5.2: The Cifar-100 dataset

Also Cifar-100, like the Cifar-10 dataset, exploits a Dirichlet distribution, parametrized by α, in order to simulate the clients' heterogeneity.

5.1.3 EMnist

The **EMnist** dataset [39] is another image classification dataset. It is composed by 814.255 images of hand-written characters split in 697.932 samples for the train set and 116.323 samples for the test set. Each image has a size of 28×28 and it is in gray-scale, meaning that it has only one color channel. EMnist has 62 classes that span all the English alphabet (lower case and upper case) plus all the digits from 0 to 9.
Image 5.3 shows some examples of images present in this dataset.

Figure 5.3: The EMnist dataset

This dataset, is not not balanced, meaning that it does not have the same number of samples for each class. Plot 5.4 shows the data distributions of the classes.

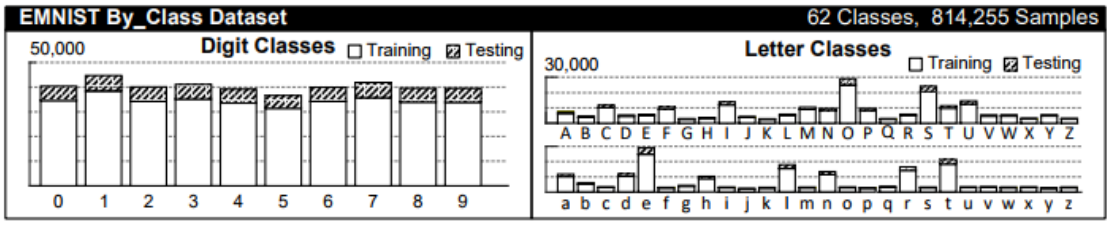

Figure 5.4: EMnist classes distributions

Regarding the federated splitting, the experiments conducted on this dataset aim to simulate the real life scenario in which each client represents a single person with its own writing style. For this reason the data are split by authorship. In total there are 3500 authors that have generated the data. Some experiments are conducted also in a balanced fashion (I.I.D), meaning that the 3500 clients have the same number of example for each class.

5.1.4 Shakespeare N.C.P

The **Shakespeare dataset** [40] belongs to the natural language domain. In particular, the task required by this dataset is *Next Character Prediction* (N.C.P): given a sub-sequence of 80 characters extracted from a sentence, the goal is to predict the 81'st. All the data are extracted from the works of the English poet and writer William Shakespeare.
In total, the dataset is composed by 200000 sub-sequences of 80 character. Each character can belong to one of the 80s classes presented in the dataset: they are all

the letters (lowercase and uppercase) plus some punctuation marks like colons and semicolons.
Regarding the federated splitting, since the sentences are retrieved from Shakespeare's plays, data are divided by character's plays. In total, 100 different character are selected, with 2000 sub-sequences of sentences each. Like the EMnist dataset, also in this case some experiments are conducted in a balanced fashion by redistributing the data inside the 100 clients such that each one has the same amount of sample for each class.

5.1.5 StackOverflow

The **StackOverflow** dataset also belongs to the natural language domain. However the task is different with respect to the Shakespeare dataset. For this dataset the task required is next word prediction, namely, given a set of words in a sentence, the goal is to predict the following one.
This dataset is a collection posts of the famous web-site *StackOverflow*. In particular 290 posts, on average, have been gathered for 342.477 users. Each post is composed by a question and a corpus. For the experiments proposed in this work, only the question part of each post is considered in order to do next word prediction.
In total, the number of different words used across all the dataset is 10.000 and, on average, each word is used 9946 times.
The federated split is made by user and all the experiments are made in a non-I.I.D. fashion since, due to the task, it has no meaning to split the data in order to create local datasets with uniform distributions.

5.2 Models used

For each dataset reported above, a particular model architecture is adopted. In this section, all the architectures chosen are reported and explained.

5.2.1 Cifar-10 and Cifar-100

Due to the similarity of these two datasets, the same model architecture is adopted for all the experiments on both Cifar-10 and Cifar-100, with the exception of the last layer that has to have an output dimension equal to the number of classes of the dataset (10 for Cifar-10 and 100 for Cifar-100).
The model architecture is called, in the literature, LeNet-5[41] and it is defined by a feature extractor, composed by a sequence of convolutional layers, and a classifier, composed by a sequence of dense (fully connected) layers.
Table 5.1 reports the full LeNet-5 architecture.

LeNet-5			
Name	**Type**	**Features**	**# parameters**
Conv1	Convolutional	kernel=5, activation=relu	64x3x5x5+64
MaxPool1	MaxPool2D	kernel=2	-
Conv2	Convolutional	kernel=5, activation=relu	64x64x5x5+64
MaxPool2	MaxPool2D	kernel=2	-
Flatten	Flatten	-	-
Fc1	Dense	activation=relu	384x1600+384
Fc2	Dense	activation=relu	192*384+192
Fc3	Dense	activation=relu	n_class*192+n_class

Table 5.1: LeNet-5 architecture. The flatten layer transforms the 2D tensor input as a 1D tensor. n_class=10 for Cifar-10, n_class=100 for Cifar-100.

5.2.2 EMnist

For the EMNIST dataset a similar CNN architecture to the previous one is used. Yet again, the network is made by two components: a feature extractor composed by a sequence of convolutional layers and then a classifier, composed by a sequence of fully connected layers.
A peculiarity of this architecture with respect to the previous one is the use of dropout layers. Each dropout layer has a parameter $p \in [0,1]$. During training time, the layer will randomly zeroes some of the elements of the input tensor with probability p. This has proven to be an effective technique for regularization and preventing the co-adaptation of neurons [Improving neural networks by preventing co-adaptation of feature detectors].
Table 5.2 reports the full architecture:

EMnist			
Name	**Type**	**Features**	**# parameters**
Conv1	Convolutional	kernel=3	32x1x3x3+32
Conv2	Convolutional	kernel=3, activation=relu	64x32x3x3+64
MaxPool1	MaxPool2D	kernel=2	-
Dropout1	Dropout	p=0.25	-
Flatten	Flatten	-	-
Fc1	Dense	-	128x9216+128
Dropout2	Dropout	p=0.5	-
Fc2	Dense	-	10x128+10

Table 5.2: EMnist network architecture. The first Convolution layer has one channel since the EMnist dataset is composed by gray-scale images.

5.2.3 Shakespeare N.C.P

For this dataset, a RNN-based architecture is used. More specifically, this architecture is composed by three elements: i) an embedding layer, ii) an LSTM module and iii) a dense layer.
The embedding layer is simply an hash-map that, for each input letter, will associate a vector of a fixed embedding size. This letters' embeddings are trainable parameters of the network, meaning that they will modify themselves during training in order to be the most suitable ones to guarantees the best network output.
The LSTM module is a particular RNN architecture proposed in [42]. It is composed internally by 2 layers of 100 neurons each. The output of the LSTM module is fed into iii), the dense layer that will output a vector of size 80 (the number of class). This output vector is then passed into a softmax function that will transform all the values of the vector into probabilities of belonging to that particular class.
Table 5.3 reports the full architecture.

Shakespeare N.C.P		
Name	**Type**	**Features**
embedding	Embedding	num_classes=80 embed_size=8
lstm	LSTM	hidden_size=100 num_layers=2
linear	Dense	-

Table 5.3: Shakespeare N.C.P architecture

5.2.4 StackOverflow

The architecture used for the StackOverflow dataset is an enhanced version of the previous one: it has the same structure composed by an embedding layer, an LSTM module and some dense layers, but the size of each one of them is much larger. The embedding layer maps each word to an embedding vector of size 96. The LSTM module is composed by one single hidden layer with 670 neurons. Instead of using one single dense layer for classification, this architecture uses two of them.
Table 5.4 shows the architecture.

StackOverflow		
Name	**Type**	**Features**
embedding	Embedding	num_classes=10004 embed_size=96
lstm	LSTM	hidden_size=670 num_layers=1
linear1	Dense	-
linear2	Dense	-

Table 5.4: StackOverflow architecture

5.3 General Analyses in FL

In this section, basic experiments on FedAvg are conducted in order to explore the basic principles of the Federated Learning paradigm. In particular, four studies are going to be conducted:

- What happens when the number of clients increase?
- What happens if the fraction of clients selected at each round changes?
- What happens if, each client, every round trains for more epochs the global model?
- What happens to the performances of the global model if the heterogeneity of the data split varies?

If not stated otherwise, the standard setting for the parameters is reported in Table 5.5

Algorithm	FedAvg
Dataset	Cifar-10
Rounds	10.000
K	100
C	0.2
E	1
α	100

Table 5.5: Standard run setting

5.3.1 Ablation on K

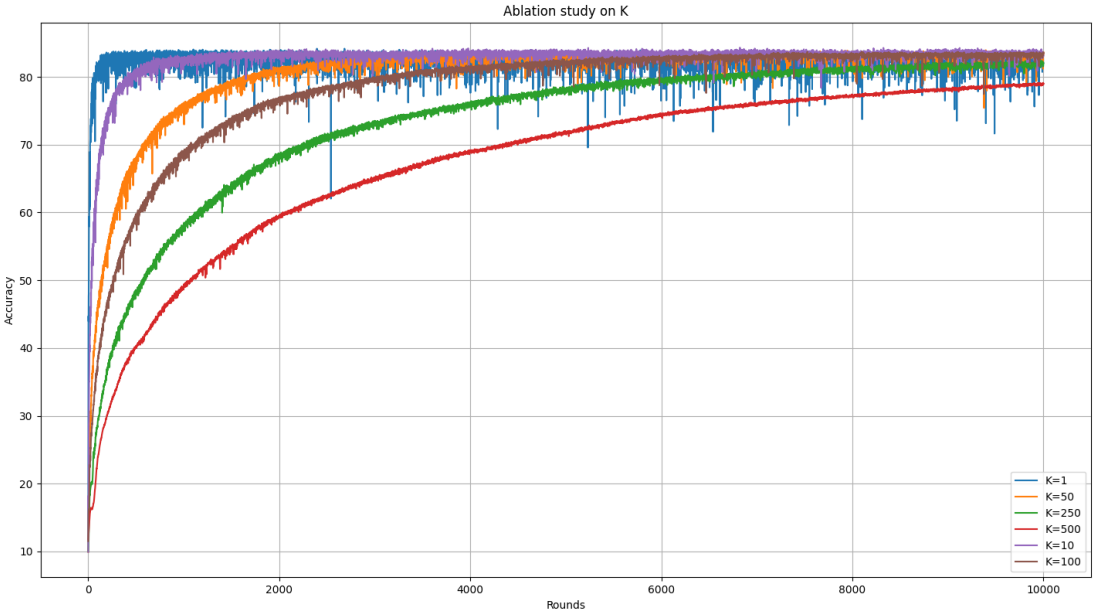

Figure 5.5: Ablation on K

In this experiment, the effect of the number of clients is studied. Figure 5.5 compares 6 different configurations in which the parameter k is, in increasing order: 1, 10, 50, 100, 250 and 500. It is immediately clear that, the smaller K, the better are the performances of the algorithm. This is due to the fact that, with a small number of clients, each one of them has a local dataset with more samples, hence, the locally trained models will be more performing. In particular, the case of $K = 1$ is equivalent to centralized learning process, since that only client will have, as local dataset, the entire data samples available. It is also important to notice that, the larger K, the more stable is the algorithm. This empirically proves that the aggregated model is less prone to have loss instability: in the $K = 500$ case, for instance, even if a single model has a performance drop in a particular round, the other 499, which represent the 99.8% of the total number of models, will mitigate the effect.

5.3.2 Ablation study on C

This ablation experiment has been conducted in order to understand what is the effect of C, the fraction of clients selected at each round. The value of C will span these different values: 0.05, 0.1, 0.2, 0.5, 0.7, 1. When $C = 1$, all clients are selected at all rounds.

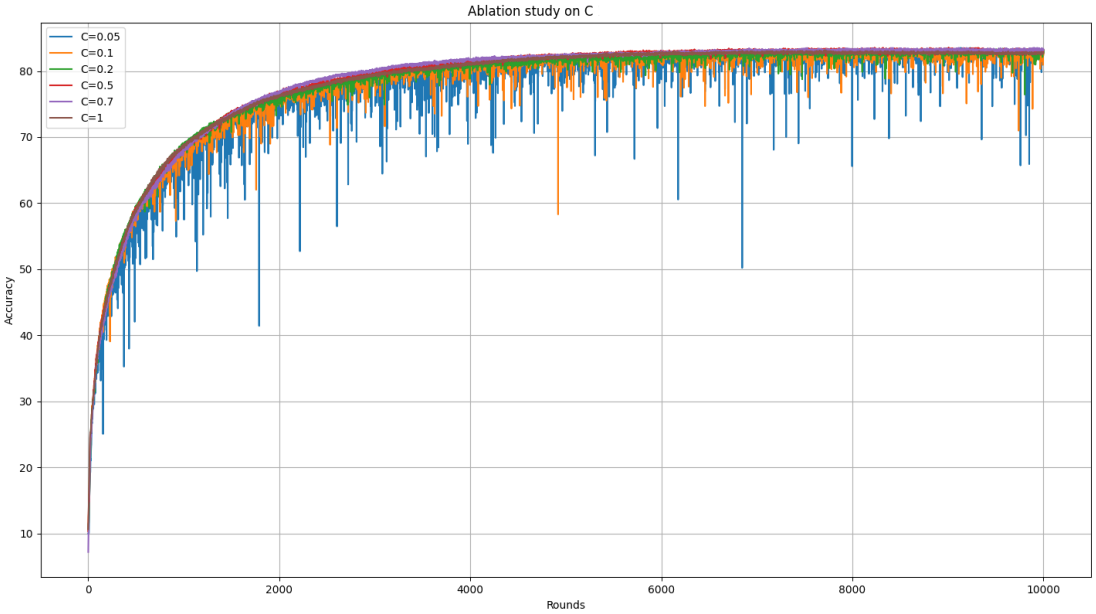

Figure 5.6: Ablation study on C

In Figure 5.6 the experiments with different configurations of the parameter C are reported: the values taken in consideration are $C = 0.05$, $C = 0.1$, $C = 0.2$, $C = 0.5$, $C = 0.7$ and $C = 1$. It is immediately clear that there are no huge differences, in terms of performances, between the different algorithms. Even from a convergence speed, all the algorithms are very comparable. The real difference stands on the noisiness of the algorithms: the smaller the C, the noisier the algorithm. This is easy explainable by the fact that, if C is small, then it is more likely that different clients are selected at each round, while, if $C = 1$ for instance, each round will have always all the clients selected and the aggregated model will be more stable.

5.3.3 Ablation study on E

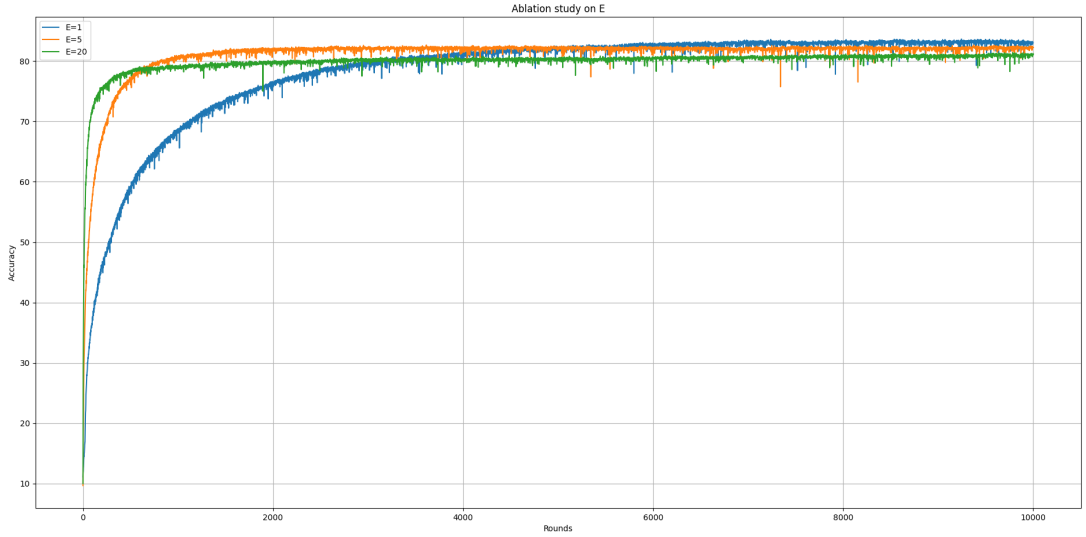

Figure 5.7: Ablation study on E.

In this experiment, the parameter E, the parameters that defines how many local epochs each selected client has to compute in order to complete a local train of a model is tested. Figure 5.7 reports four different configurations in which E is equal to 1, 5 and 20. It is important to remember that, in an FL scenario, client computational resources are limited, hence, a too large value of E could be unfeasible. Moreover, in an FL scenario, clients' local datasets are generally small, which means that a too large value of E could lead to a heavily overfitted model. From Figure 5.7, it is clear that, the smaller E, the slower is the convergence of the algorithm w.r.t the accuracy. It is worth to mention that this convergence slowdown effect is less effective when E is sufficiently large: the difference in convergence between the setups in which $E = 5$ and $E = 1$ can reach 10 points in accuracy, especially in the first rounds. On the other hand, the difference between the setups $E = 20$ and $E = 5$ is way smaller. Lastly, it is clear that when $E = 20$ the algorithm reaches worse overall performances, even though, in the first rounds the accuracy curve is very steep. This is due to the fact that the local models are trained for too many epochs on clients data, hence they are overfitted, which leads too poor global performances.

5.3.4 Ablation on α

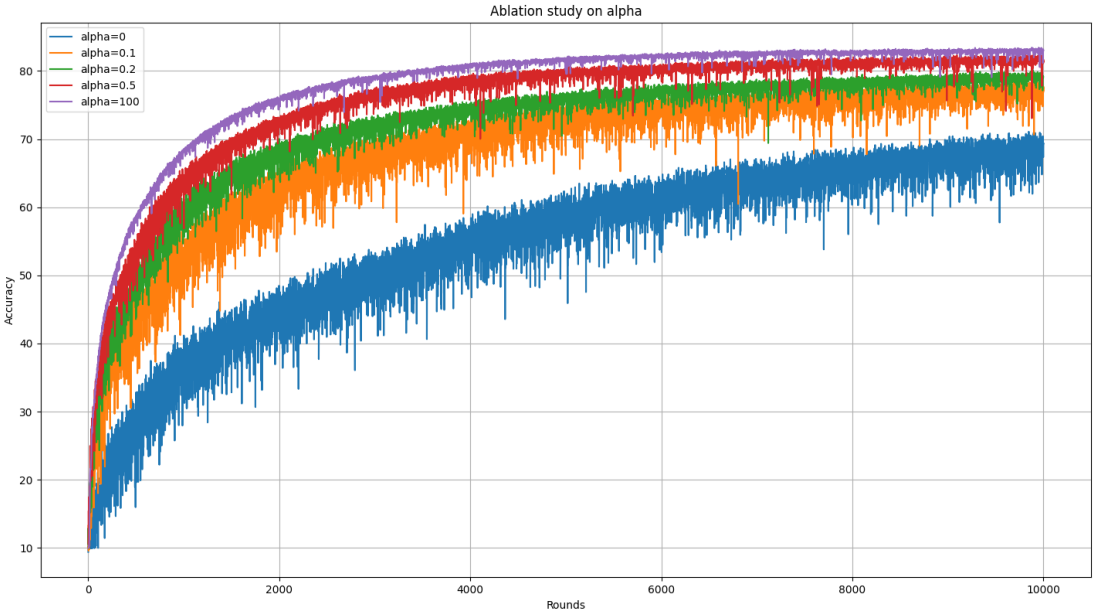

Figure 5.8: Ablation on α

In this experiment, the effect of α has been tested. In particular, four different values of α has been evaluated: from the most heterogeneous scenario of $\alpha = 0$, in which each client will have only data who belong to one class, to the most homogeneous one, $\alpha = 100$, in which each client will have a uniform local data distribution among all classes. From Figure 5.8, it is clear that the higher the α, the more performing is the algorithm. Moreover, it is interesting to notice that, the smaller the α, the noisier the algorithm behaviour. Both of these effects are justified by the fact that, local trained models tend to be more similar if they are trained on similar local datasets, hence, if α is big, each client will have a local dataset with a similar distribution w.r.t the other ones. It is important to notice that the difference between a small-α and a big-α scenario is not only in the accuracy reached but also in the speed of convergence: in a big-α scenario, even after 2000 rounds, the algorithm is almost on its convergence, while, for $\alpha = 0$, the accuracy curve after round 2000 is still very steep.

5.4 FedSeq vs S.O.T.A

In this section all the experiments that puts in comparison FedSeq with the other FL state of the art algorithms are presented. The S.O.T.A algorithms involved in this

analysis are: FedAvg, SCAFFOLD, FedProx and FedDyn. All the experiments are conducted on the datasets presented in Section 5.1. Moreover, for each dataset, the analyses are proposed with different values of selected clients per round $C \in [0.1,0.2]$ and in different settings of data distribution (IID and NIID, or $\alpha \in [0.1,0.2,0.5]$ for the Cifar datasets).

5.4.1 Shakespeare dataset

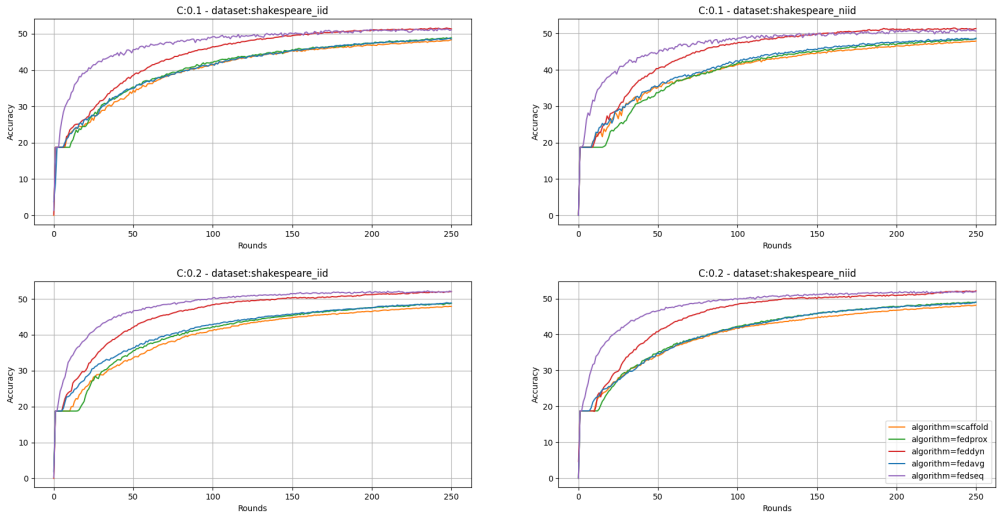

Figure 5.9: Comparison of FL algorithms on Shakespeare dataset

Figure 5.9 shows all the experiments conducted on the Shakespeare dataset, for 250 rounds, in four different configurations of C and IID/NIID data distribution. It is clear that FedSeq is the fastest algorithm from a convergence point of view, regardless the configurations of C and the dataset IIDness. Generally FedDyn is the only algorithm capable of reaching the FedSeq accuracies. On average, FedAvg, SCAFFOLD and FedProx have worse performances reaching 3-4% less points in accuracy w.r.t FedSeq and FedDyn. It is important to remember that each algorithm presented in Figure 5.9 uses the best combination of parameters that each algorithm requires (found with an extensive grid-search):

- $C = 0.1$, IID configuration: **FedSeq** uses ψ_{conf}, ϕ_{greedy}, $\tau = Kullback$, the maximum number of client per superclient is set to 5 and the minimum number of examples is 8000. **FedDyn** sets $\alpha = 0.015$, while **FedProx** sets $\mu = 0.0001$.
- $C = 0.1$, NIID configuration: **FedSeq** uses ψ_{clf}, ϕ_{greedy}, $\tau = Wasserstein$, the maximum number of client per superclient is set to 5 and the minimum

number of examples is 8000. **FedDyn** sets $\alpha = 0.001$, while **FedProx** sets $\mu = 0.0001$.

- $C = 0.2$, IID configuration: **FedSeq** uses ψ_{conf}, ϕ_{greedy}, $\tau = Gini$, the maximum number of client per superclient is set to 5 and the minimum number of examples is 8000. **FedDyn** sets $\alpha = 0.001$, while **FedProx** sets $\mu = 0.0001$.

- $C = 0.2$, NIID configuration: **FedSeq** uses ψ_{clf}, ϕ_{greedy}, $\tau = Wasserstein$, the maximum number of client per superclient is set to 5 and the minimum number of examples is 8000. **FedDyn** sets $\alpha = 0.001$, while **FedProx** sets $\mu = 0.001$.

5.4.2 EMnist dataset

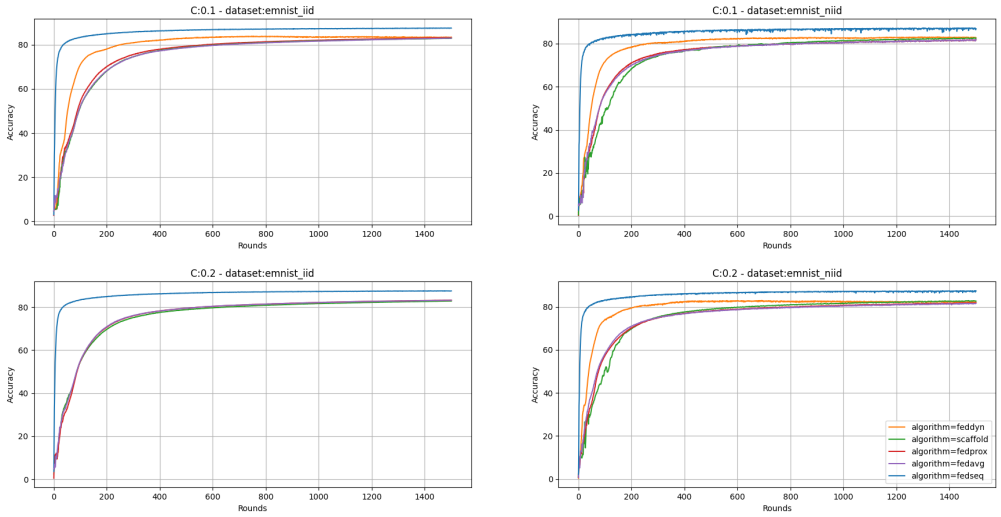

Figure 5.10: Comparison of FL algorithms on EMnist dataset

Figure 5.10 shows all the experiments conducted on the EMnist dataset, for 1500 rounds, in four different configurations of C and IID/NIID data distribution. Also with this dataset, regardless the configuration, FedSeq outperforms the other algorithms in both accuracy reached and speed of convergence. FedDyn shows a better speed of convergence w.r.t the other S.O.T.A algorithms even though the final accuracy reached is the same: 3-4 percentage point lower w.r.t the accuracy achieved by FedSeq. For this dataset, the parameters chosen for each algorithm are now reported:

- $C = 0.1$, IID configuration: **FedSeq** uses ψ_{clf}, ϕ_{KMeans}, $\tau = Euclidean$, the maximum number of client per superclient is set to 21 and the minimum number of examples is 4120. **FedDyn** sets $\alpha = 0.0155$, while **FedProx** sets $\mu = 0.0001$.
- $C = 0.1$, NIID configuration: **FedSeq** uses ψ_{clf}, ϕ_{greedy}, $\tau = Cosine$, the maximum number of client per superclient is set to 21 and the minimum number of examples is 4120. **FedDyn** sets $\alpha = 0.001$, while **FedProx** sets $\mu = 0.0001$.
- $C = 0.2$, IID configuration: **FedSeq** uses ψ_{conf}, ϕ_{KMeans}, $\tau = Euclidean$, the maximum number of client per superclient is set to 21 and the minimum number of examples is 4120. **FedDyn** is missing in this setup due to lack of resources, while **FedProx** sets $\mu = 0.001$.
- $C = 0.2$, IID configuration: **FedSeq** uses ψ_{conf}, ϕ_{KMeans}, $\tau = Euclidean$, the maximum number of client per superclient is set to 21 and the minimum number of examples is 4120. **FedDyn** sets $\alpha = 0.01$, while **FedProx** sets $\mu = 0.001$.

5.4.3 Cifar10 dataset

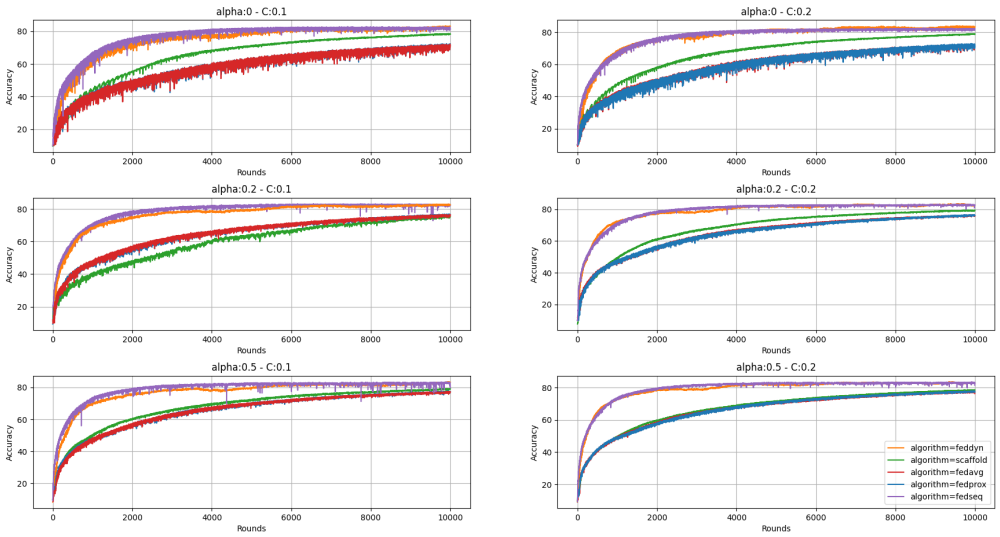

Figure 5.11: Comparison of FL algorithms on Cifar-10 dataset

Figure 5.1 shows six different experiments conducted on the Cifar-10 dataset. The value for C varies from 0.1 and 0.2 while α can assume three different configurations

0, 0.2 and 0.5. Regardless the configuration, FedSeq reaches better results in both terms of accuracy and speed of convergence. The only algorithm that is competitive is FedDyn, which is capable of reaching the same performances of FedSeq. FedAvg and FedProx perform similarly while SCAFFOLD is better especially when α is small, showing that it performs better when the scenario tends to be more NIID. In the following, the parameters chosen for each algorithm are now reported:

- $C = 0.1$, $\alpha = 0$ configuration: **FedSeq** uses ψ_{conf}, ϕ_{greedy}, $\tau = Kullback$, the maximum number of client per superclient is set to 11 and the minimum number of examples is 800. **FedDyn** sets $\alpha = 0.001$, while **FedProx** sets $\mu = 0.0001$.

- $C = 0.1$, $\alpha = 0.2$ configuration: **FedSeq** uses ψ_{conf}, ϕ_{greedy}, $\tau = Kullback$, the maximum number of client per superclient is set to 11 and the minimum number of examples is 800. **FedDyn** sets $\alpha = 0.01$, while **FedProx** sets $\mu = 0.001$.

- $C = 0.1$, $\alpha = 0.5$ configuration: **FedSeq** uses ψ_{clf}, ϕ_{greedy}, $\tau = Cosine$, the maximum number of client per superclient is set to 11 and the minimum number of examples is 800. **FedDyn** sets $\alpha = 0.015$, while **FedProx** sets $\mu = 0.0001$.

- $C = 0.2$, $\alpha = 0$ configuration: **FedSeq** uses ψ_{conf}, ϕ_{greedy}, $\tau = kullback$, the maximum number of client per superclient is set to 11 and the minimum number of examples is 800. **FedDyn** sets $\alpha = 0.001$, while **FedProx** sets $\mu = 0.01$.

- $C = 0.2$, $\alpha = 0.2$ configuration: **FedSeq** uses ψ_{conf}, ϕ_{greedy}, $\tau = Kullback$, the maximum number of client per superclient is set to 11 and the minimum number of examples is 800. **FedDyn** sets $\alpha = 0.001$, while **FedProx** sets $\mu = 0.01$.

- $C = 0.2$, $\alpha = 0.5$ configuration: **FedSeq** uses ψ_{clf}, ϕ_{greedy}, $\tau = Cosine$, the maximum number of client per superclient is set to 11 and the minimum number of examples is 800. **FedDyn** sets $\alpha = 0.001$, while **FedProx** sets $\mu = 0.01$.

5.4.4 Cifar100 dataset

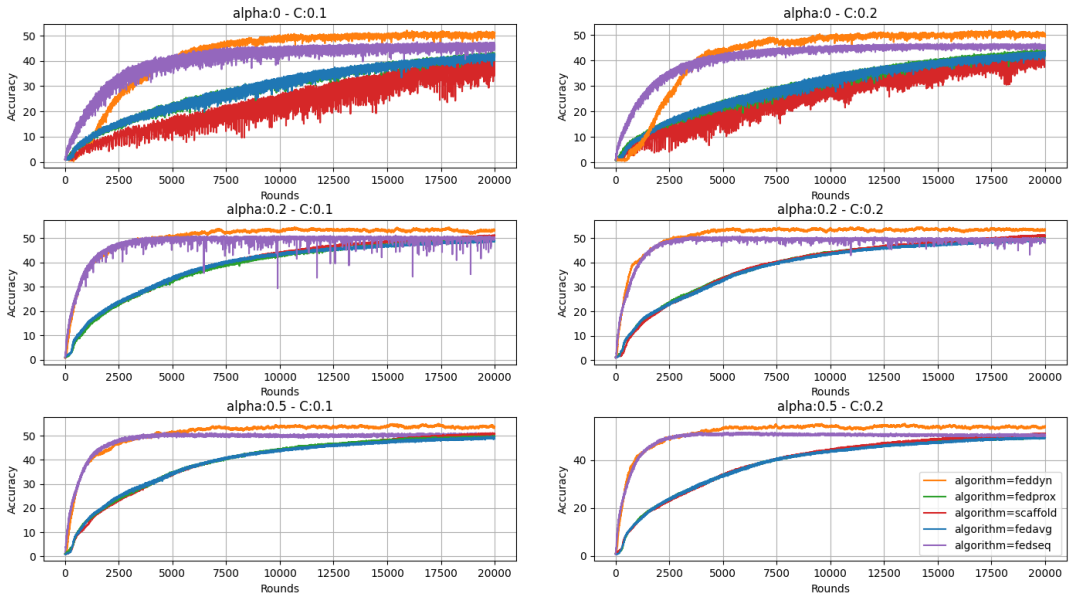

Figure 5.12: Comparison of FL algorithms on Cifar-100 dataset

Figure 5.12 shows six different experiments conducted on the Cifar-100 dataset. The value for C varies between 0.1 and 0.2 while α can assume three different configurations 0, 0.2 and 0.5. It is immediately clear that FedSeq and FedDyn are the best algorithms: they are able to reach higher accuracies than the others (up to 8% in certain configurations) while being very fast in convergence (in most of the cases, after 2500 rounds they reach stability). FedAvg, FedProx and SCAFFOLD show a similar behaviour in every configuration. Another aspect to consider is the noisiness of the algorithm: it is clear that, when $\alpha = 0$, every experiment is noisier. SCAFFOLD seems to be the most susceptible in this regard.
In the following, the parameters chosen for each algorithm are now reported:

- $C = 0.1$, $\alpha = 0$ configuration: **FedSeq** uses ψ_{conf}, ϕ_{greedy}, $\tau = Kullback$, the maximum number of client per superclient is set to 11 and the minimum number of examples is 800. **FedDyn** sets $\alpha = 0.001$, while **FedProx** sets $\mu = 0.001$.

- $C = 0.1$, $\alpha = 0.2$ configuration: **FedSeq** uses ψ_{conf}, ϕ_{greedy}, $\tau = Kullback$, the maximum number of client per superclient is set to 11 and the minimum number of examples is 800. **FedDyn** sets $\alpha = 0.001$, while **FedProx** sets $\mu = 0.0001$.

- $C = 0.1$, $\alpha = 0.5$ configuration: **FedSeq** uses ψ_{conf}, ϕ_{greedy}, $\tau = Kullback$, the maximum number of client per superclient is set to 11 and the minimum number of examples is 800. **FedDyn** sets $\alpha = 0.01$, while **FedProx** sets $\mu = 0.001$.

- $C = 0.2$, $\alpha = 0$ configuration: **FedSeq** uses ψ_{conf}, ϕ_{greedy}, $\tau = Kullback$, the maximum number of client per superclient is set to 11 and the minimum number of examples is 800. **FedDyn** sets $\alpha = 0.015$, while **FedProx** sets $\mu = 0.01$.

- $C = 0.2$, $\alpha = 0.2$ configuration: **FedSeq** uses ψ_{conf}, ϕ_{greedy}, $\tau = Kullback$, the maximum number of client per superclient is set to 11 and the minimum number of examples is 800. **FedDyn** sets $\alpha = 0.015$, while **FedProx** sets $\mu = 0.01$.

- $C = 0.2$, $\alpha = 0.5$ configuration: **FedSeq** uses ψ_{conf}, ϕ_{greedy}, $\tau = Kullback$, the maximum number of client per superclient is set to 11 and the minimum number of examples is 800. **FedDyn** sets $\alpha = 0.015$, while **FedProx** sets $\mu = 0.01$.

5.4.5 StackOverflow dataset

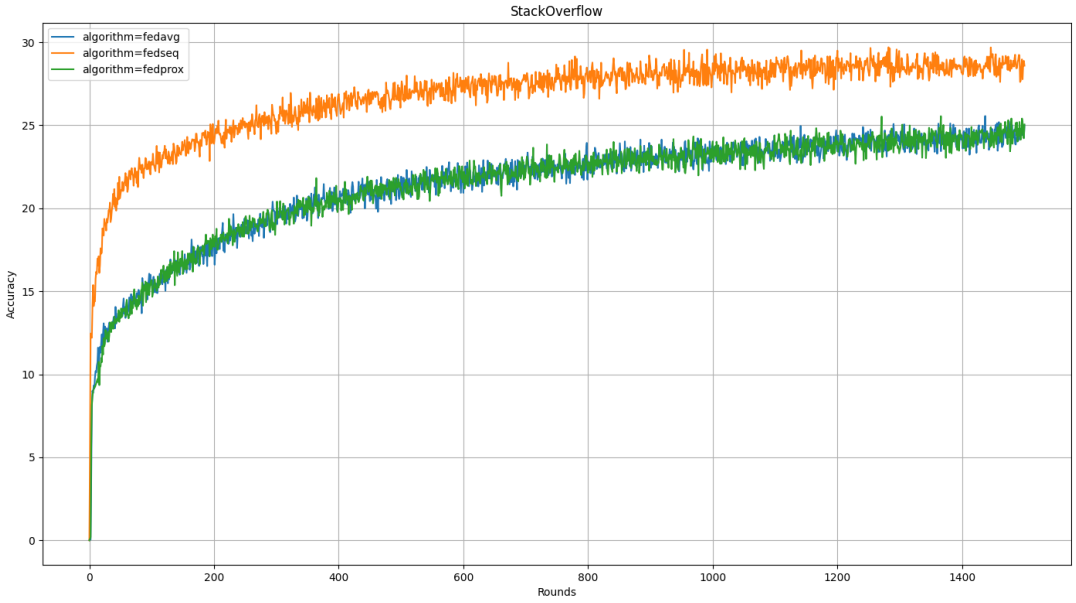

Figure 5.13: Comparison of FL algorithms on StackOverflow dataset

Figure 5.13 shows the results related to the NLP StackOverflow dataset. Due to its huge size, the number of experiments had to be reduced: in particular, C was set to 0.01 reducing the number of selected clients per round to 400. Moreover FedDyn and SCAFFOLD require more resources in order to be executed, hence it was not possible to conduct the experiments for those algorithms with the StackOverflow dataset. Regarding the experiments reported in Figure 5.13, it is clear that FedSeq is able of reaching a better performances w.r.t FedProx and FedAvg gaining, on average, 4 more points in accuracy.
For this dataset, the parameters chosen for each algorithm are now reported:

- **FedSeq** uses ψ_{conf}, ϕ_{random} the maximum number of client per superclient is set to 25 and the minimum number of examples is 8500.

- **FedProx** sets $\mu = 0.0001$.

5.5 Ablation on FedSeq

ψ	ϕ	τ	$\alpha = 0$	$\alpha = 0.2$	$\alpha = 0.5$
		Cifar-10			
-	random	-	81.90	82.09	82.12
clf	K-means	Euclidean	**82.30**	81.78	82.48
conf	K-means	Euclidean	82.04	81.99	82.37
conf	greedy	KL	82.21	**82.20**	82.22
conf	greedy	Cosine	82.09	81.85	82.71
clf	greedy	Cosine	79.95	82.06	**82.83**
		Cifar-100			
-	random	-	**46.39**	48.62	49.44
clf	K-means	Euclidean	44.91	48.74	49.60
conf	K-means	Euclidean	43.55	49.43	49.79
conf	greedy	KL	45.97	**49.56**	**49.82**
conf	greedy	Cosine	45.79	48.98	49.61
clf	greedy	Cosine	45.22	48.92	49.62

Table 5.6: FedSeq baselines: comparison of grouping criteria by varying ϕ, ψ and τ. Results in terms of accuracy (%).

Table 5.6 reports an ablation study conducted on FedSeq for its parameters client approximator ψ and grouping method ϕ. For the sake of simplicity, all the experiments involve only the Cifar datasets (Cifar10 and Cifar100). The best performances are reported in bold.
The differences in terms of accuracy between the runs are not very important. Even the random method is competitive with the other setups.
When $\alpha = 0.5$ (a more IID scenario), we can notice a slight improvement by all the configurations from the random setup: this is due to the fact that, in this scenario, clients are likely to have the majority of the classes in their local datasets, hence, the algorithm must be able to cluster together those clients who have complementary classes in their local data.
The same reason, explains why, in the scenario Cifar-100 $\alpha = 0$, the configuration that reaches the best performances is the random one: since every client has only one class (among the 100 considered in Cifar-100), it is very likely to cluster together clients with different classes, even if the cluster strategy is a random one.

5.6 Privacy attacks

In this section two different attacks are presented and then tested: the GAN attack[16] and the label flipping attack[43]. In particular, the goal of the following experiments, is to empirically test the resilience of FedSeq against the aforementioned attacks. To do so, all the experiments are conducted with the FedSeq and FedAvg algorithms, in order to understand if the former is more resilient with respect to the latter.

5.6.1 The GAN attack

The GAN attack, proposed in [16] is a *passive* FL attack that aims at reconstructing private data. In order to accomplish that, at round t, an attacker $a \in [C_i]_{i=1}^{K}$, disguised as a client, exploits the incoming trained model θ_t as discriminator inside a GAN[14] architecture. More specifically, when the attacker a receives the model θ_t from the server, it trains, like every client, the model on its local data, but it also stores locally the model θ_t. The attacker now creates a GAN architecture $\Lambda(G, D)$ composed by a generator G and a discriminator $D = \theta_t$.
The main idea behind this reconstruction attack follows the fact that θ_t has gained some knowledge about the past $t - 1$ rounds of training on other clients. If the attacker a is capable of training a GAN model Λ such that its generator G is capable of "fooling" θ_t into classifying with high probability a generated image, such reconstruction must resemble data on which θ_t has been trained on.
In the original work[16], the last classification layer of θ_t was changed in size with the addition of another class, dedicated to the classification of fake images

constructed by the generator G. Moreover, it is assumed that the attacker a is selected every round. Since those two assumptions significantly deviate from a real life scenario, in these experiments different assumptions must be made:

1. The incoming model θ_t must remain invariant. For this reason, just at the end of the architecture, a dense layer with an output shape of one is added. This is done in order to have a Discriminator capable of binary classifies an incoming input into "real" or "fake" without modifying the original model's architecture.

2. Just like every other client, the attacker a is selected in only a fraction of the total rounds. This means that the incoming model θ_t has been trained for an unknown (for the attacker) number of rounds on an unknown number of clients.

3. The architecture and the parameters of the Generator follows [44].

Evaluation of the attack

In order to evaluate how "good" is the GAN reconstruction, the metric classically used in the literature is the **Fréchet inception distance** (F.I.D) metric, introduced in [45]. This distance measures how far are two multivariate Gaussian distributions $X_1 \sim (\mu_1, \sigma_1)$ and $X_2 \sim (\mu_2, \sigma_2)$:

$$FID(X_1, X_2) = |\mu_1 - \mu_2|^2 + Tr(\sigma_1 + \sigma_2 - 2\sqrt{\sigma_1 \cdot \sigma_2}) \tag{5.2}$$

The F.I.D is calculated by assuming that X_1 and X_2 are the outputs of the coding layer $pool_3$ of a pre-trained *InceptionV3* model (see below) for generated samples and real world samples respectively. μ_n is the mean and σ_n the covariance of the outputs of the pre-trained network over all real world or generated samples. The **lower** the F.I.D, the **better** the reconstruction.

FedAvg and FedSeq comparison

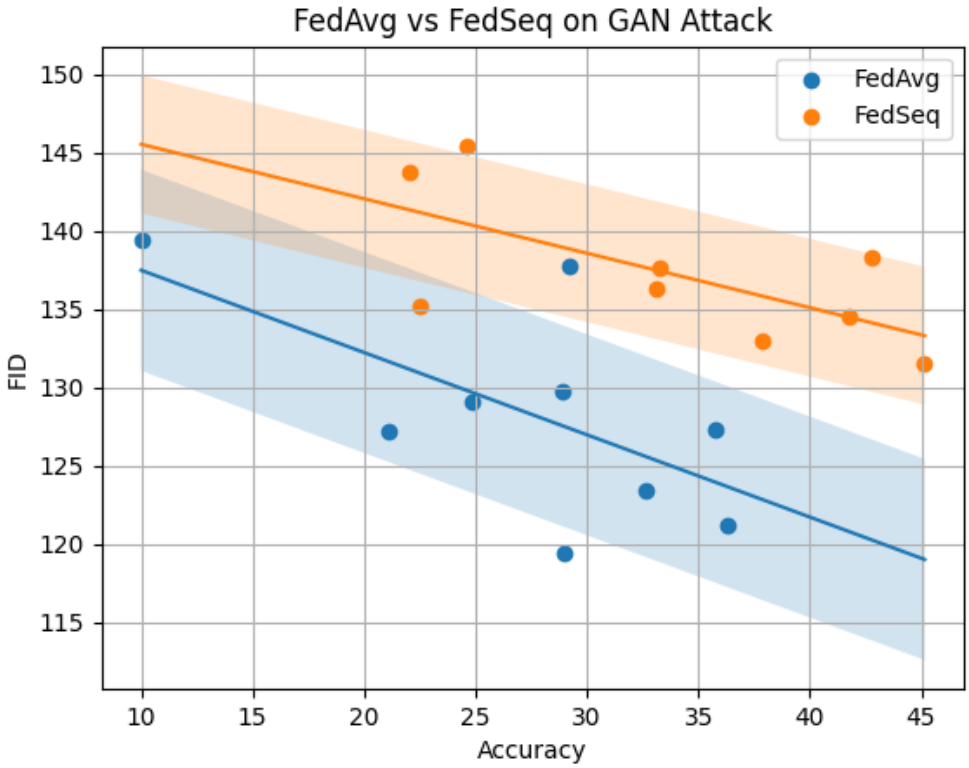

Figure 5.14: FedAvg vs FedSeq on GAN Attack

Figure 5.14 reports different reconstructions F.I.D metrics conducted on different levels of accuracy for the discriminator on the dataset Cifar-10. It is clear that, on average, the F.I.D metric is lower when the algorithm used is FedAvg, hence for the attacker it is easier to steal private informations from other clients if the FL algorithm used is FedAvg. This means that FedSeq, other that obtaining better performances, it is also more resilient against this attack.

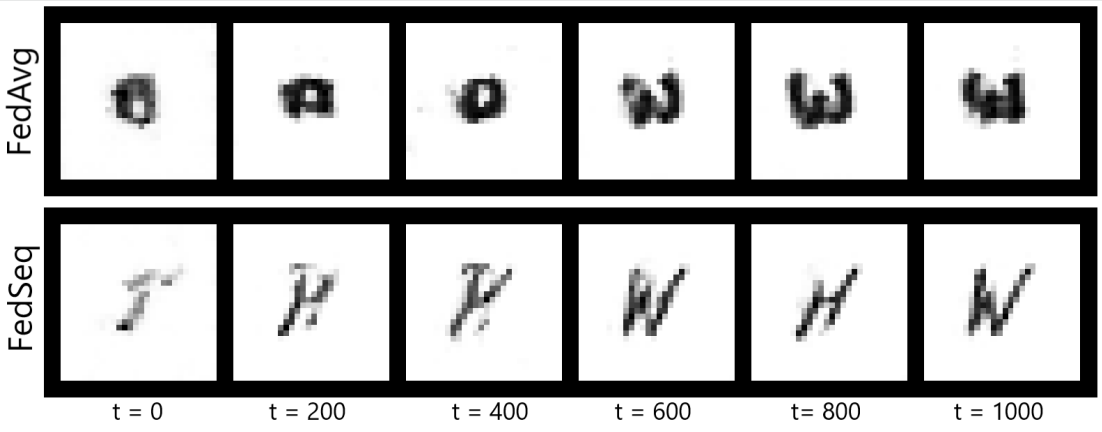

Figure 5.15: Some examples of reconstruction for FedAvg and FedSeq on the EMNIST dataset

Figure 5.15 shows some examples of reconstruction for both the algorithms FedAvg and FedSeq. It is important to notice that the images are reconstructed at a specific round t. Since FedSeq reaches good performances way before FedAvg, the discriminator used for its reconstruction is significantly better. Nevertheless, the FedSeq reconstructions quality is not higher that the FedAvg reconstructions. This qualitatively proves that FedSeq is more resilient to that attack with respect to FedAvg.

5.6.2 The Label Flipping attack

The Label Flipping attack, introduced in [43], is an *active* privacy attack that aims at deteriorating the FL algorithm performances. The idea behind this attack is the following: a set $[a_i]_{i=1}^{LK} \subseteq [C_i]_{i=1}^{K},\ L \in [0,1]$ of clients, willingly swaps the labels of their local data, through a set of criteria $\{\gamma_i\}_{i=1}^{LK}$ in order to mislead the global model classification ability. Each criterion γ_i defines, for the attacker i, which labels will be swapped during the attack. All the experiments aimed at testing this attack are done over the Cifar-10 and Cifar-100 datasets.

Experiments setup

Different experiments have been conducted in order to test the resilience of FedAvg and FedSeq against this attack.
The fraction L of attackers varies among $\{0.1, 0.3, 0.5\}$. For FedSeq a fraction L of superclients are considered attackers: inside those superclients all the clients are attackers.
Each experiment is also divided in *random* or *fixed*: when an experiment is *random*,

each criterion γ_i is defined independently from the other criteria γ_j $j \neq i$, meaning that different attackers are going to swap different classes. On the other hand, when the experiment is *fixed*, $\gamma_i = \gamma_j$ $\forall i, j$, meaning that all the attackers are going to swap the same classes.
Regarding the classes swapped, it is important to distinguish between the two datasets involved. For Cifar-10, three types of class swapping are proposed: following [43], class **0** (airplane) and class **2** (birds) or class **5** (dog) and class **3** (cat) are swapped. A configuration in which all the four aforementioned classes (0,2,3,5) are paired and swapped is also proposed.
For Cifar-100 two configurations of class swapping are proposed: swapping 20 classes that do not belong to the same superclass (meaning one class for each superclass) and swapping 20 classes but pairing only those classes that belong to the same superclass. The first configuration will be referenced as **extra_SC** while the second one **intra_SC**.
All the experiments are ran for only 1000 rounds, with $\alpha = 100$ and $K = 100$ clients on Cifar-10 and Cifar-100 datasets.

algorithm	dataset	swapped classes	method	global acc.	swapped acc.
FedAvg	Cifar-10	0-2	Fixed	64.79	51.43
		5-3	Fixed	65.93	41.50
		0-2-5-3	Fixed	62.56	47.97
			Random	64.56	52.75
	Cifar-100	extra_SC	Fixed	26.90	17.50
			Random	26.40	19.05
		intra_SC	Fixed	26.83	21.89
			Random	27.39	24.33
FedSeq	Cifar-10	0-2	Fixed	72,69	61,16
		5-3	Fixed	74,66	52,54
		0-2-5-3	Fixed	70,19	53,47
			Random	72,18	58,57
	Cifar-100	extra_SC	Fixed	36,89	26,46
			Random	37,22	26,80
		intra_SC	Fixed	37,80	30,65
			Random	38,30	32,04

Table 5.7: Label Flipping experiments averaged by fraction of attackers

Table 5.7 reports, averaged by L, both the global accuracies and the accuracies computed on only the swapped classes for each configuration attack. It is clear that the swapped accuracy is consistently lower that the global one, meaning that the global model struggles in correctly classify the attacked classes. Moreover, the more the number of attacked classes, the lower the global accuracies. On average, the *fixed* attacks are more effective than the *random* ones. For CIFAR-100, there is a

significant difference between **extra_SC** and **intra_SC** configurations, resulting in more effective attacks when the swapped classes do not belong to the same superclass.

L (%)	global acc.	swapped acc.
Cifar-10		
0%	8.59	-
10%	8.74	6.62
30%	7.68	9.86
50%	7.5	7.6
Cifar-100		
0%	11.22	-
10%	11.22	9.67
30%	10.50	8.60
50%	9.66	8.43

Table 5.8: Each cell represents the difference in performances between FedSeq and FedAvg by averaging all the attack configurations.

Table 5.8 shows the difference in accuracies, both the global and the swapped ones, between FedSeq and FedAvg by averaging all the experiments results. The results are presented by fraction of attackers and dataset. It is clear that, the larger L, the thinner the difference in performances between the two algorithms. Even though FedSeq seems to be less resilient to the attack, even when $L = 0.5$, the algorithm performs significantly better than FedAvg.

Chapter 6

Conclusion

In this work of thesis the Federated Learning world was explored. Initially, a theoretical formulation of the FL learning approach was formulated and put in comparison with the traditional centralized one.
Different problems arise within the FL approach. Among them, one of the most important is the *heterogeneity* problem: since a FL training process is executed in a decentralized fashion, data are split in K local datasets stored in K different devices and, for privacy reasons, they cannot be shared. The *heterogeneity* problem emerges when different local datasets have different data distributions, forcing the local trained models to be incompatible for a proper aggregation that can produce a performing aggregated model.
FedAvg [5] was the first algorithm in the FL scenario and it is not able of handling the *heterogeneity* problem. In this work, a non-exhaustive taxonomy of the current state of the art algorithms in FL that try to solve the aforementioned problem has been analyzed. Among them it is possible to cite FedProx [6], FedDyn [17] and SCAFFOLD [18].
The main part of this thesis has been dedicated to FedSeq[7], a novel FL algorithm that, by clustering together dissimilar clients in order to get clusters with homogeneous data distributions, is capable of training local models in the clusters following a sequential approach.
Chapter 4 is dedicated to the formal definition of FedSeq and all of its components: the different *client approximators* and *grouping methods* coupled with different *metrics*.
Chapter 5 is dedicated to the experiments: first, different general analysis of the behaviour of different FL parameters like the number of clients of the clients' data distributions has been conducted. Then a deep ablation analysis dedicated to the parameters of FedSeq. Lastly a comparison between FedSeq and other state of the art algorithms has been conducted: these experiments were made on five different datasets in two different tasks: image recognition and NLP prediction.

The results shows that, overall, FedSeq is capable of surpassing (or, at least matching), the other algorithms in every configuration, regardless the task, the dataset or the client's data distributions. From a speed of convergence point of view, FedSeq shows great improvements with respect to the other algorithms, even reaching a 7x reduction in certain configurations.
Another important aspect of this thesis is the privacy in FL: section 3.3 is dedicated to a non-exhaustive taxonomy of different privacy attacks and defences proposed in the literature. Among the attacks, the *passive* GAN attack proposed in [16] and the *active* Label Flipping attack proposed in [43] have been tested against FedSeq and FedAvg. The results show that FedSeq, even if capable of reaching better performances, is not more susceptible to those attacks, meaning that its improvements in performances don't come at a cost in terms of privacy.

Bibliography

[1] PN Druzhkov and VD Kustikova. «A survey of deep learning methods and software tools for image classification and object detection». In: *Pattern Recognition and Image Analysis* 26.1 (2016), pp. 9–15 (cit. on p. 1).

[2] Abdullah-Al Nahid and Yinan Kong. «Involvement of machine learning for breast cancer image classification: a survey». In: *Computational and mathematical methods in medicine* 2017 (2017) (cit. on p. 1).

[3] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. «Supervised learning». In: *Machine learning techniques for multimedia*. Springer, 2008, pp. 21–49 (cit. on p. 1).

[4] *2018 reform of EU data protection rules*. European Commission. May 25, 2018. URL: `https://ec.europa.eu/commission/sites/beta-political/files/data-protection-factsheet-changes_en.pdf` (visited on 06/17/2019) (cit. on p. 1).

[5] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. «Communication-efficient learning of deep networks from decentralized data». In: *Artificial intelligence and statistics*. PMLR. 2017, pp. 1273–1282 (cit. on pp. 2, 20, 21, 62).

[6] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. «Federated learning: Challenges, methods, and future directions». In: *IEEE Signal Processing Magazine* 37.3 (2020), pp. 50–60 (cit. on pp. 2, 21, 62).

[7] Riccardo Zaccone, Andrea Rizzardi, Debora Caldarola, Marco Ciccone, and Barbara Caputo. «Speeding up Heterogeneous Federated Learning with Sequentially Trained Superclients». In: *arXiv preprint arXiv:2201.10899* (2022) (cit. on pp. 2, 26, 30, 62).

[8] Yann LeCun, Yoshua Bengio, et al. «Convolutional networks for images, speech, and time series». In: *The handbook of brain theory and neural networks* 3361.10 (1995), p. 1995 (cit. on pp. 4, 12).

[9] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. «Deep learning». In: *nature* 521.7553 (2015), pp. 436–444 (cit. on p. 4).

[10] Kenneth Ward Church. «Word2Vec». In: *Natural Language Engineering* 23.1 (2017), pp. 155–162 (cit. on p. 5).

[11] Frank Rosenblatt. «The perceptron: a probabilistic model for information storage and organization in the brain.» In: *Psychological review* 65.6 (1958), p. 386 (cit. on p. 5).

[12] Robert Hecht-Nielsen. «Theory of the backpropagation neural network». In: *Neural networks for perception*. Elsevier, 1992, pp. 65–93 (cit. on p. 7).

[13] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985 (cit. on p. 15).

[14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. «Generative adversarial networks». In: *Communications of the ACM* 63.11 (2020), pp. 139–144 (cit. on pp. 16, 56).

[15] John Nash Jr. «Non-cooperative games». In: *Essays on Game Theory*. Edward Elgar Publishing, 1996, pp. 22–33 (cit. on p. 16).

[16] Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. «Deep models under the GAN: information leakage from collaborative deep learning». In: *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2017, pp. 603–618 (cit. on pp. 17, 56, 63).

[17] Durmus Alp Emre Acar, Yue Zhao, Ramon Matas Navarro, Matthew Mattina, Paul N Whatmough, and Venkatesh Saligrama. «Federated learning based on dynamic regularization». In: *arXiv preprint arXiv:2111.04263* (2021) (cit. on pp. 21, 62).

[18] Sai Praneeth Karimireddy, Satyen Kale, Mehryar Mohri, Sashank Reddi, Sebastian Stich, and Ananda Theertha Suresh. «Scaffold: Stochastic controlled averaging for federated learning». In: *International Conference on Machine Learning*. PMLR. 2020, pp. 5132–5143 (cit. on pp. 21, 62).

[19] Ming Xie, Guodong Long, Tao Shen, Tianyi Zhou, Xianzhi Wang, Jing Jiang, and Chengqi Zhang. «Multi-center federated learning». In: *arXiv preprint arXiv:2005.01026* (2020) (cit. on pp. 24, 25).

[20] Felix Sattler, Klaus-Robert Müller, and Wojciech Samek. «Clustered federated learning: Model-agnostic distributed multitask optimization under privacy constraints». In: *IEEE transactions on neural networks and learning systems* 32.8 (2020), pp. 3710–3722 (cit. on p. 25).

[21] J MacQueen. «Classification and analysis of multivariate observations». In: *5th Berkeley Symp. Math. Statist. Probability*. 1967, pp. 281–297 (cit. on pp. 25, 35).

[22] Xuefei Yin, Yanming Zhu, and Jiankun Hu. «A comprehensive survey of privacy-preserving federated learning: A taxonomy, review, and future directions». In: *ACM Computing Surveys (CSUR)* 54.6 (2021), pp. 1–36 (cit. on pp. 27, 28).

[23] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. «On data banks and privacy homomorphisms». In: *Foundations of secure computation* 4.11 (1978), pp. 169–180 (cit. on p. 28).

[24] Yoshinori Aono, Takuya Hayashi, Lihua Wang, Shiho Moriai, et al. «Privacy-preserving deep learning via additively homomorphic encryption». In: *IEEE Transactions on Information Forensics and Security* 13.5 (2017), pp. 1333–1345 (cit. on p. 28).

[25] Adi Shamir. «How to share a secret». In: *Communications of the ACM* 22.11 (1979), pp. 612–613 (cit. on p. 28).

[26] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. «Practical secure aggregation for privacy-preserving machine learning». In: *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1175–1191 (cit. on p. 28).

[27] Payman Mohassel and Yupeng Zhang. «Secureml: A system for scalable privacy-preserving machine learning». In: *2017 IEEE symposium on security and privacy (SP)*. IEEE. 2017, pp. 19–38 (cit. on p. 28).

[28] Cynthia Dwork, Aaron Roth, et al. «The algorithmic foundations of differential privacy». In: *Foundations and Trends® in Theoretical Computer Science* 9.3–4 (2014), pp. 211–407 (cit. on p. 28).

[29] Graham Cormode, Somesh Jha, Tejas Kulkarni, Ninghui Li, Divesh Srivastava, and Tianhao Wang. «Privacy at scale: Local differential privacy in practice». In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 1655–1658 (cit. on p. 29).

[30] Dakshi Agrawal and Charu C Aggarwal. «On the design and quantification of privacy preserving data mining algorithms». In: *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2001, pp. 247–255 (cit. on p. 29).

[31] Keke Chen and Ling Liu. «A survey of multiplicative perturbation for privacy-preserving data mining». In: *Privacy-Preserving Data Mining*. Springer, 2008, pp. 157–181 (cit. on p. 29).

[32] Pierangela Samarati and Latanya Sweeney. «Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression». In: (1998) (cit. on p. 29).

[33] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. «l-diversity: Privacy beyond k-anonymity». In: *ACM Transactions on Knowledge Discovery from Data (TKDD)* 1.1 (2007), 3–es (cit. on p. 29).

[34] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. «t-closeness: Privacy beyond k-anonymity and l-diversity». In: *2007 IEEE 23rd international conference on data engineering*. IEEE. 2006, pp. 106–115 (cit. on p. 29).

[35] Alessandro Achille, Michael Lam, Rahul Tewari, Avinash Ravichandran, Subhransu Maji, Charless C Fowlkes, Stefano Soatto, and Pietro Perona. «Task2vec: Task embedding for meta-learning». In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2019, pp. 6430–6439 (cit. on p. 32).

[36] Hervé Abdi and Lynne J Williams. «Principal component analysis». In: *Wiley interdisciplinary reviews: computational statistics* 2.4 (2010), pp. 433–459 (cit. on p. 32).

[37] Alex Krizhevsky, Geoffrey Hinton, et al. «Learning multiple layers of features from tiny images». In: (2009) (cit. on pp. 37, 38).

[38] Tzu-Ming Harry Hsu, Hang Qi, and Matthew Brown. «Measuring the effects of non-identical data distribution for federated visual classification». In: *arXiv preprint arXiv:1909.06335* (2019) (cit. on p. 38).

[39] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. «EMNIST: Extending MNIST to handwritten letters». In: *2017 international joint conference on neural networks (IJCNN)*. IEEE. 2017, pp. 2921–2926 (cit. on p. 39).

[40] Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečn, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. «Leaf: A benchmark for federated settings». In: *arXiv preprint arXiv:1812.01097* (2018) (cit. on p. 40).

[41] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324 (cit. on p. 41).

[42] Sepp Hochreiter and Jürgen Schmidhuber. «Long short-term memory». In: *Neural computation* 9.8 (1997), pp. 1735–1780 (cit. on p. 43).

[43] Vale Tolpegin, Stacey Truex, Mehmet Emre Gursoy, and Ling Liu. «Data poisoning attacks against federated learning systems». In: *European Symposium on Research in Computer Security*. Springer. 2020, pp. 480–501 (cit. on pp. 56, 59, 60, 63).

[44] Alec Radford, Luke Metz, and Soumith Chintala. «Unsupervised representation learning with deep convolutional generative adversarial networks». In: *arXiv preprint arXiv:1511.06434* (2015) (cit. on p. 57).

[45] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. «GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium». In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/8a1d694707eb0fefe65871369074926d-Paper.pdf (cit. on p. 57).