POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Matematica
Dipartimento di Scienze Matematiche

Combining Split and Federated Architectures for Efficiency and Privacy in Deep Learning

Tesi di Laurea di:
Valeria Turina

Relatore: Prof. Guido Marchetto
Relatore Esterno: Prof. Flavio Esposito

Acknowledgements

This thesis has been conducted in the Department of Computer Science at Saint Louis University. The thesis has been partially supported by the National Science Foundation, award CNS: #1908574.
Special gratitude goes out to Professor Flavio Esposito that allowed me the opportunity to spend an amazing period at the Saint Louis University in Missouri and followed me in each moment of this experience abroad and to Professor Guido Marchetto for the precious help from Italy.
A sincere thanks to Francesco, for being close to me, despite the distance, during the last months, and for all the support over these years.
I would like to thank my parents, my family, and my friends to provide me a continuous encouragement throughout my years of study and during this experience in the US.
I am grateful to Emily, Sweta, Matteo, and the two Roberto, that were my US family in Saint Louis and were able to alleviate the difficulties encountered during this pandemic period.
Thanks to Chiara, a perfect adventure companion, roommate, and friend.
Finally, a special thought goes to Irene for the great lesson of life that she has taught me.

Table of Contents

List of Figures

List of Tables

Abstract

Mobile phones and wearable devices are used and carried everywhere by people, producing every day a large amount of distributed and sensitive data.
Classical machine learning approaches collect such data on usually on a single machine to compute training models and obtain useful predictions.
To better preserve user and data privacy and at the same time guarantee high performance, distributed machine learning techniques such as Federated and Split Learning have been recently proposed.
This work tries to improve the efficiency and privacy of Split and Federated learning combining them in two different types of architectures using the PySyft library implemented inside PyTorch. The goal is trying to reduce the computational power requested from each client by *Federated Learning* and to parallelize the *Split Learning* avoiding problems with unbalanced dataset.
The code has been developed to easily switch from local to remote learning, simulating the distributed process inside the same machine and, then, running the neural network on workers located on different devices.
Then, a trade-off analysis between the two architectures in terms of efficiency and privacy was performed: changing the distribution of data inside each device, estimating the possibility to rebuild the original data using the neural network inversion attack, and exploring possible privacy improvements.
Finally, the results encourage us to further investigate the architectures and find which ones are more suitable to maximize both the performance and privacy, according to the dataset.

Chapter 1

Introduction

The use of mobile phones and wearable devices produces a vast amount of data that can be useful to improve the user experience if applied to text or speech recognition or to support the doctors in preventing diseases such as heart attack.

Standard machine learning algorithms are based on a centralized approach in which the training is performed on a single machine after collecting data coming from different devices. One of the most critical issues of such architectures is that the data used for training is usually privacy sensitive.
These facts, combined with the increasing computational power of mobile devices, lead McMahan et al. [1] to introduce the notion of Federated Learning in 2016. Federated Learning allows training of a neural network without sharing the raw training data, that hence can be privately owned by each process (e.g., a device or an agent) participating in the training. The principle requires sending the model to the training processes and then an aggregation of the different neural network weights is exchanged during the training phase.
This approach became popular even among large companies. For example, Google uses Federated Learning to improve the Gboard mobile keyboard [2] [3], and Apple is using Federated learning to obtain better performance in the intelligent assistant "Hey Siri." Furthermore, Federated Learning has proven to be useful also for other applications that require training over privacy-sensitive distributed systems, such as for medical imaging or medical data analysis [4].

In 2018 a new distributed paradigm was studied to train a deep neural network using multiple data sources without sharing raw data. The authors, a team of MIT researchers, called this approach *splitNN* [5]: the model is split between two or more machines, and only the intermediate results of the neural network, computed by the previous worker, are shared. *SplitNN* lowered the computing power required at each device to train the neural network with respect to a classical federated learning model.

While Federated Learning needs devices with large computational power, the Split Learning cannot be computed in a parallel way and it could not arrive at convergence with a really unbalanced dataset.

In this work, carried out during an exchange program in the Department of Computer Science at Saint Louis University in Missouri, we study how to improve the efficiency and privacy of Split and Federated learning, combining them in two different types of architectures. Our implementation uses the PySyft library [6] implemented within the PyTorch framework.

We chose that library because (currently) it allows greater versatility and freedom than other libraries for distributed training, such as TensorFlow Federated [7]. We solved several machine learning design and implementation challenges. In particular, to design our novel combined split and federated learning architectures, we had to decompose the computational dynamic graph created by Pytorch when an operation is executed; then using the VirtualWorkers API of PySyft [6] the two architectures were implemented simulating the presence of different machines on a single laptop.
We evaluate the performance of our split and federated learning architectures with respect to efficiency and privacy. In particular, we explored the impact of different distributions of data between clients, and evaluated under which condition the architectures preserve privacy even under neural network inversion attacks.
We tested our algorithms on real world datasets, such as MNIST and the Covid-19 CT images of chest dataset where the study of privacy is critical.
Finally, the results encourage us to further investigate the architectures and find which ones are more suitable to maximize both the performance and the privacy, according to the dataset.

The rest of the thesis is organized as follows. In Chapter 2, we discuss some background on distributed learning, while in the following we describe the Parallel [8] and the Federated Split Learning: two architectures that inherits some properties from Split and Federated Learning. In Chapter 4, after an explanation of the Library *PySyft* [6], we show a performance analysis of the two architectures based on the dataset MNIST. A privacy attack to these two distributed approaches is explained proposing a possible solution in Chapter 5. Finally in Chapter 6 we show the results obtained using the dataset Covid-19.

Chapter 2

Background and Related Work on Distributed Learning

In this chapter, we present a brief introduction of distributed learning algorithms. With the term *Distributed Learning Algorithms*, we define a machine learning algorithm in which the training phase is performed from many distributed devices without the need of sharing private data.

First of all, we discuss the *Federated Learning* and the *Split Learning* algorithm, two of the most popular distributed techniques, that we try to join together to improve their weaknesses. Then, the chapter follows with a description of two techniques, recently studied, that combine the previous two methods.

2.1 Federated Learning

We describe the *Federated Learning* algorithm to study how we can improve this method. In particular, one of the most popular problems is that it requires distributed devices to have a lot of computational power and to be able to train the entire model.
For this reason, joining this method with the *Split Learning* can avoid this problem.

In a centralized system, training and evaluation of a model are computed on cloud data collected from different devices and stored in a cloud. After the validation step, the cloud or the server sends the final model to devices to compute the model inference.

On the other hand, Federated Learning can perform training and evaluate decentralized data without storing sensitive information on the same machine, preserving training data privacy.

The different devices merely exchange small updates, necessary to compute the global training of the model [1].

Notation
K : numbers of clients
w : model parameters
$l(x_i, y_i; w)$: loss function
$\mathcal{P}_k$**:** set of indexes for each local dataset
n_k**:** $|\mathcal{P}_k|$
C : fraction of clients selected at each training round
K : total number of clients
$\mathbf{R}^d$: set of real numbers of dimension d
η : learning rate

Objective function:[1]

$$\min_{w \in \mathbf{R}^d} f(w)$$

$$s.t. \quad f(w) = \frac{1}{n} \sum_{k=1}^{K} n_k F_k(w)$$

$$F_k(w) = \frac{1}{n_k} \sum_{i \in \mathcal{P}_k} f_i(w)$$

If the partition of data made by $\mathcal{P}_k$ is uniform the function $f(w)$ is the expectation over $\mathcal{P}_k$ of $F_k(w)$. An example of function $f_i(w)$ could be the loss function $l(x_i, y_i; w)$, where x_i and y_i represent the set of data.

FederatedSGD Algorithm

This algorithm is based on the federated optimization, and uses the Stochastic Gradient Descent to compute the final aggregate result. During each round, all the clients K are selected. Each client computes the gradient $\nabla F_k(w_t)$ on its local dataset. Then the server computes the average gradient and updates the parameter w_t. At the beginning of the following step, it sends the updated parameter to each client[1].

FederatedAveraging Algorithm

This algorithm represents an improvement of the method described above. In the Federated Averaging Algorithm, each client updates locally the parameter w, and the server computes the average of the updated parameters.

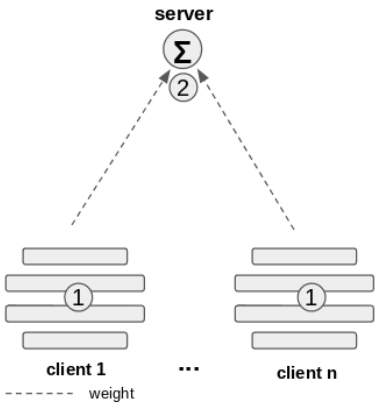

Figure 2.1: Federated Learning Forward Propagation

Algorithm 1 Federated Averaging [1]

The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate.

Server executes:

initialize w_0

for each round t = 1,2,... **do**
  $m \leftarrow \max(CK,1)$
  $S_t \leftarrow$(random set of m clients)
  for each clients $k \in S_t$ in parallel **do**
    $w_{t+1} \leftarrow$ClientUpdate(k, w)
  end for
  $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$
end for

ClientUpdate(k, m):

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

Synchronous and Asynchronous Federated Learning

An important aspect of Federated Learning is the difference between Synchronous and Asynchronous methods to merge the information coming from different clients. The table 2.1 lists some differences of these two approaches.

Synchronous	**Asynchronous**
It assumes that all devices can always send the information and at the same time	Connection between devices could not always be possible (i.e. if the device is uncharged)
Less realistic	More realistic
Time of transmission is constant	Time of transmission is not constant
It needs synchronized clocks to manage the goal	It do not need a mechanism to synchronize processes
The clients' updates can be sent to the secure server at job completion.	The clients' updates can be send to the secure server continuously,
Each client should compute his updates more or less in the same time, otherwise there could be a delay for all the algorithm	Each client can compute his updates anytime and can send to the server when finished

Table 2.1: Difference between Synchronous and Asynchronous Federated Learning

Possible problems with Federated Learning

The Federated Learning optimization problem was characterized by the following aspects [1]:

- Non-IID dataset: each client collects data based on his personal use of the device. Therefore, the client's dataset could not be representative of the population distribution.
  The data of each client can vary a lot in term of size collected for each different types of labels (in this case, we have an unbalanced local dataset); in term of statistical distribution for each client (for example, for an image classification task each node can contain different kinds of data if they are situated in a different part of the world).

- Massively distributed: the number of clients used in training is larger than the average number of examples in each local dataset.

- Limited communication: some devices could be offline or discharged during the training rounds.

- Communication cost: each client sends, at each round, the updates to a secure server that computes the average. Even if sending weights or gradients to a server is better in terms of privacy than sending the data directly, this could be expensive in communication. If the machine learning used is a deep neural

network, the number of parameters to send to the server could also be around 40 million data for each client.

2.2 Private Aggregation of Teacher Ensembles (PATE) Algorithm

Another federated learning algorithm that was recently published with a different type of model aggregation (or model averaging) is known as *PATE*, Private Aggregation of Teacher Ensembles [9]. The idea is that there are *n "teachers"* able to train each model on their disjoint datasets, and at the end, they all reach a consensus given a new dataset of input never seen before and owned by a *Student*. This algorithm is useful when the Student's data is unlabeled; in fact, sending the data to each Teacher, we can obtain the prediction on the Student's data without sharing the other dataset.

The algorithm can be divided in three main parts [9]:

1. Ensemble of n teacher models: each teacher train a set of data that is a part of the global partition. They can use different techniques to obtain a trained model able to solve a specific task. Then working similarly to other ensemble models such as Random Forest, each teacher computes the label for the *Student's* data.

2. Aggregation mechanism: in the case in which the n *Teachers* reached a strong consensus, the labels predicted do not depend on a specific *Teacher* model. The aggregation mechanism consists of the count of votes assigned by each *Teacher* to each class. Furthermore, to increase privacy during this phase, Gaussian noise was added to the final aggregated votes.

3. Student model: finally PATE is able to compute the output labels of *Student's* unlabeled data using the *Teacher's* knowledge.

One of the most significant drawbacks of this approach is that *Student's* data has to be sent to each *Teacher* to be labeled. This data exchange may be unfeasible if, for example, the *Student* is a hospital that is not allowed to send their sensitive data to other parties for patient privacy law enforcement.
A variation of the original PATE algorithm was proposed to solve the data privacy problem: PATE Bidirectionally Private. The latter encrypts the Student's data to guarantee confidentiality, using, for example, an Additive Secret Sharing algorithm.

2.3 Split Learning

We describe the *Split Learning* algorithm, which does not require large computational power for each device. A drawback of this approach is the sequential training between the clients. In fact, the server can work with just a single client at a time. This fact can also lead to non-convergence of the neural network in the case of dealing with very unbalanced clients' dataset.

In 2018, Otkrist Gupta and Ramesh Raskar [5] [10] proposed a new idea to train a deep neural network using different data located in different sources (called Alice(s)) without sharing data with a supercomputer (called Bob) that helps data entities to obtain the result.
In one of the more straightforward Split Learning configurations, each client (Alice) trains a deep neural network up to a specific layer, called *"cut layer"*, and then Bob receives the client's output and completes the training, despite having no access to the raw data. The backpropagation phase of the deep neural network follows a similar approach: the gradient is backpropagated from the last layer computed by Bob to the *"cut layer"*, and then Alice completes the computation with her layers [5].

In the rest of this section, we first describe the Split Learning algorithm using a single agent and a supercomputer , and then an improvement of this method where multiple agents are used for training.

Distributed training over a single agent

One of the design goals of this split learning algorithm [5] is to produce the same output of its centralized "unsplit learning" counterpart. The forward and backward functions are not changed but merely divided into two blocks to run in separate clients.

Notation:
N : total number of layer of a deep neural network
F_a : Alice's forward function
L_i, $i = 1, \ldots, n$: Alice's layers of deep neural network
F_b : Bob's forward function
L_i, $i = n+1, \ldots, N$: Bob's layers of deep neural network
$Send(X, Y)$: function able to sent X to Y
ϕ : random inizialiation
F_a^T : Alice's backward function
F_b^T : Bob's backward function
G' : loss function and gradient
$gradient$: gradient variable
$gradient'$: gradient variable after back propagation
F'_a : Alice's output of backward propagation after updating weights.

Algorithm 2 Distributed training over single agent [10]

initialize:
$\phi \leftarrow$ Random Initializer (e.g. Gaussian)
$F_a \leftarrow L_0, L_1, \dots L_n$
$F_b \leftarrow L_{n+1}, L_{n+2}, \dots L_N$
Alice randomly initializes the weights of F_a using ϕ
Bob randomly initializes the weights of F_b using ϕ
  while *Alice* has new data to train on **do**
    $X \leftarrow F_a(data)$
    $Send((X, label), Bob)$
    $output \leftarrow F_b(X)$
    $gradient \leftarrow G'(output, label)$
    $gradient' \leftarrow F_b^T(\text{gradient})$
    $Send(gradient', Alice)$
    $F_a' \leftarrow F_a^T(gradient')$
  end while

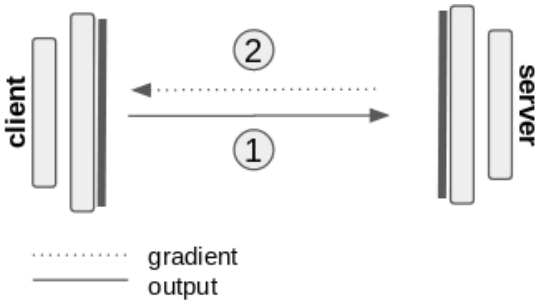

Figure 2.2: Split Learning Algorithm

After a random initialization of the weights, the client Alice computes the forward propagation and sends the output of her last layer of the neural network to the supercomputer (Bob) that completes the forward propagation and computes the gradients. Finally, the gradients computed are sent back to Alice that updates the deep neural network's weights.

Note that the labels are sent to Bob at the beginning of each training phase, to compute the loss function and backpropagate the gradients. This method cannot be applied in applications in which sending labels means sending private information.

To improve this aspect, a group of MIT researchers [10] proposed an improvement that consisted of sending the output of Bob's activation function to Alice, who owns the labels and can compute the loss function. This algorithm was called *U* shape splitNN since the shape of the layers and data workflow recalls the letter *U*.

Distributed training over multiple agents

Notation:

$F_{a,j}$: $Alice'_j s$ forward function
$F^T_{a,j}$: $Alice' s$ backward function
d : number of clients ($Alice$)
$j = 1,\ldots, d$: index for each $Alice$

In the algorithm below we present the **Distribute training over multiple agent** [5]

Algorithm 3 Distributed training over multiple agent [10]

initialize:
$\phi \leftarrow$ Random Initializer (e.g. Gaussian)
$F_{a,1} \leftarrow L_0, L_1, \ldots L_n$
$F_b \leftarrow L_{n+1}, L_{n+2}, \ldots L_N$
$Alice_1$ randomly initializes the weights of $F_{a,1}$ using ϕ
Bob randomly initializes the weights of F_b using ϕ
Bob sets $Alice_1$ as last trained

while Bob waits for Alice$_j$ to send data **do**
  $Alice_j$ requests Bob for last $Alice_o$ that trained
  $F_{a,j} \leftarrow F_{a,o}$
  $X \leftarrow F_{a,j}(data)$
  $Send((X, label), Bob)$
  $output \leftarrow F_b(X)$
  $gradient \leftarrow G'(output, label)$
  $gradient' \leftarrow F^T_b$(gradient)
  $Send(gradient', Alice_j)$
  $F'_{a,j} \leftarrow F^T_{a,j}(gradient')$
  Bob sets $Alice_j$ as last trained
end while

This architecture [10] is similar to the previous one, but instead of one client, a set of multiple agents create a distributed algorithm. This approach exploits different data owned by the clients in a safer way than the standard centralized training. In this algorithm, the training is performed sequentially, and each client

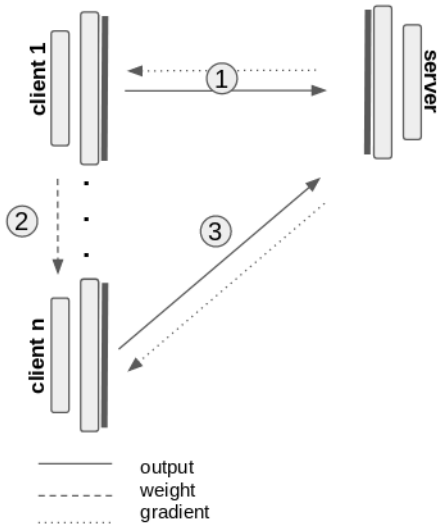

Figure 2.3: Split Learning Sequantial Algorithm

receives the last updates on the model weights from the server. Then the training between Alice and Bob is equal to the *Distributed training over single agent* [5] with the only difference that at the end of the backward phase, the updated weights are sent to the server Bob who sends them to the next client or directly from a client to the next one.

Remark. *In the following chapter, this architecture was called Sequential Split Learning to distinguish it from other architectures, described below, that try to parallelize the distributed learning.*

2.4 Combining Split and Federated Learning

In this last section, we discuss two distributed techniques that merge *Split Learning* and *Federated Learning*.
They are the *Parallel Split Learning* [8] and *SplitFed Learning* [11]. We describe the *Parallel Split Learning* in the following chapter because this is one of the two architectures that we have implemented and analyzed.

SplitFed Learning

The *SplitFed Learning* [11] is a distributed algorithm that merges the idea of computing the average, characteristic of *Federated Learning*, and the neural network split between client and server as in the *Split Learning*.

Each client computes the forward propagation, then the result of this computation is sent to the server that finishes the forward propagation. The basic

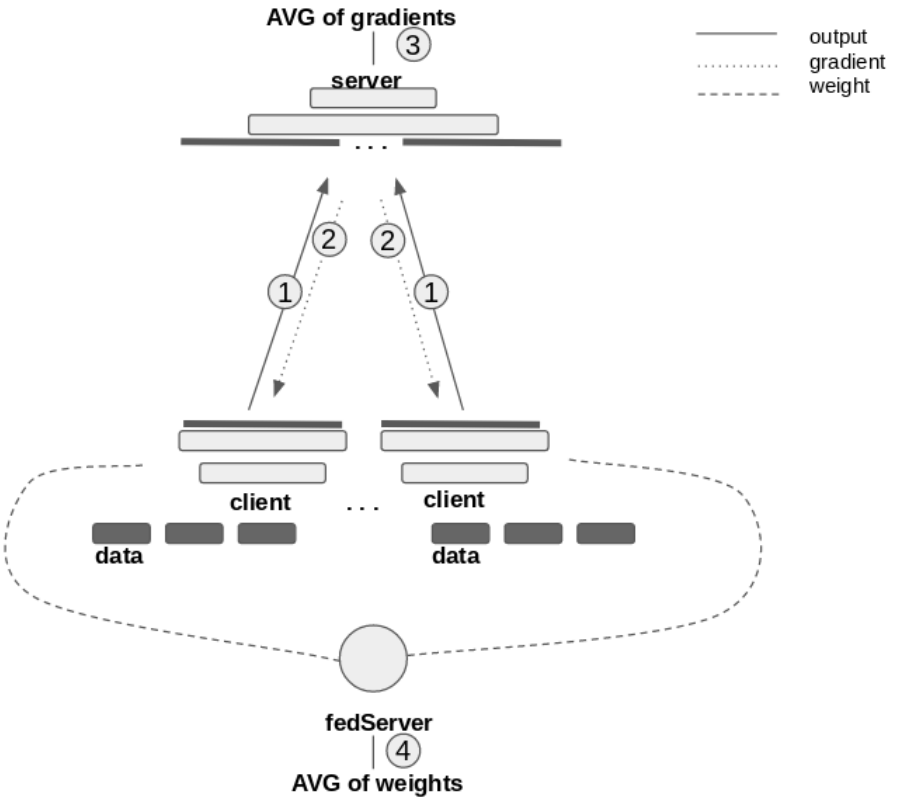

Figure 2.4: SplitFed Algorithm

version of this algorithm computes this last part using a parallel approach and then computes an average of all clients' gradients. A subsequent version of this part of the algorithm computes the forward propagation of the server's neural network sequentially, choosing randomly the clients' output to use.

The backward propagation is computed inside the server, and then the result is backward propagated inside each client. Before updating the clients' weights, a secure server, called *FedServer*, computes the average of the clients' weights and then sends back to each client the result of this operation.

Chapter 3

Split and Federated Learning Architectures

In this section, we present other two different types of distributed algorithms. They inherit some properties from Split Learning and Federated Learning. As other distributed private algorithms, they allow training of a model using data from different clients to guarantee a higher privacy level. Furthermore, compared to the distributed models previously described, they try to combine the positive aspects of Split and Federated Learning to improve their weaknesses.
In fact, in Federated Learning, each client must train the entire model; for this reason, great computational power on each node is required.
On the other hand, in Federated Learning, each client sends a secure server the model *weights*. The Split Neural Network instead shares the data modified by the first layers computed in the client. To compensate for the privacy loss due to this property, Split Learning architectures often require the addition of cryptographic methods or new techniques to prevent privacy attacks, such as reverse engineering of the neural network to obtain the original data.
To solve this problem, researchers [5, 12] proposed a few possible improvements: using a fully connected layer, or modifying the loss function in each client to maximize the difference between the output of the activation function in the cut layer and the input data.
One of the most critical aspects of these two architectures is the parallelization of the training phase, impossible in the *Sequential Split Learning*. In the latter, the first client has to finish the training with the server before we can send an updated model to the following client to continue the training phase.

Notation
N : total number of layer of a deep neural network
ϕ: Random layer inizializer
X: array of X_i, $Alice_i$'s activation output of the cut layer.
$F_{a,i}$: $Alice_i$'s forward function
L_i, $i = 1, \ldots, n$: Alice's layers of deep neural network
F_b: Bob's forward function
L_i, $i = n+1, \ldots, N$: Bob's layers of deep neural network
$Send(X, Y)$: function able to sent X to Y
G' : loss function and gradient
$gradient$: gradient variable
$gradient'$: gradient variable after back propagation
$F'_{a,i}$: $Alice_i$'s output of backward propagation after updating weights.
I : set of clients.
$|.|$: cardinality of a set.
n : number of layers to send to each client.

3.1 Parallel Split Learning

Parallel Split Learning [8] consists of $|I|$ different clients that own the first part of the model and another worker, usually a big server, that has the second part of the neural network. We do not compute the average of the gradients before updating each client's weights, as described in [8], because we assume that the client's part should remain different for each client who can own different distribution of data. First, we initialize the weights of the first part of the neural network to the same values for each client and the server. Then, for each epoch and each set of a batch of data, characterized with the same size, we send data to each client and labels to the big server, called Bob.
The forward propagation is computed for each client, and the intermediate activation output is sent to the server that continues the forward and computes the loss function using a batch size equal to $|I| * (client's \quad batch)$.
After this step, the gradient of the loss is computed and backpropagated among Bob's layers. Finally, each client receives the same updated gradient and carries out the last step of the backpropagation. An important aspect to consider is that each client must send Bob the same number of mini-batches because the server's mini-batch size is fixed to $|I| * (client's \quad batch)$.
Furthermore, this is a synchronous algorithm. We can modify it in an asynchronous architecture using batch size different for each client. The averaging operation was computed by a secure worker.

Algorithm 4 ParallelSplit [8]

initialize:

$\phi \leftarrow$ Random Initializer (e.g. Gaussian)

$F_{a,i} \leftarrow L_0, L_1, \dots L_n, \quad \forall \quad i \quad in \quad I$

$F_b \leftarrow L_{n+1}, L_{n+2}, \dots L_N$

$Alice_i$ randomly initializes the weights of $F_{a,i}$ using ϕ, $\forall$ i in I

Bob randomly initializes the weights of F_b using ϕ

for *epoch* in *epochs* **do**:

while *batches*, *labels* in trainloader **do**

send $batch \in batches$ to $Alice_i$, $\forall$ i in I

send *labels* to Bob

$X_i \leftarrow F_{a,i}(batch)$ $\forall i$ in I

$Send((X_i, labels), Bob)$

$output \leftarrow F_b(X)$

$gradient \leftarrow G'(output, labels)$

$gradient' \leftarrow F_b^T(gradient)$

Send($gradient'$, $Alice_i$), $\forall i$ in I

$F'_{a,i} \leftarrow F_{a,i}^T(gradient')$, $\forall i$ in I

end while

end for

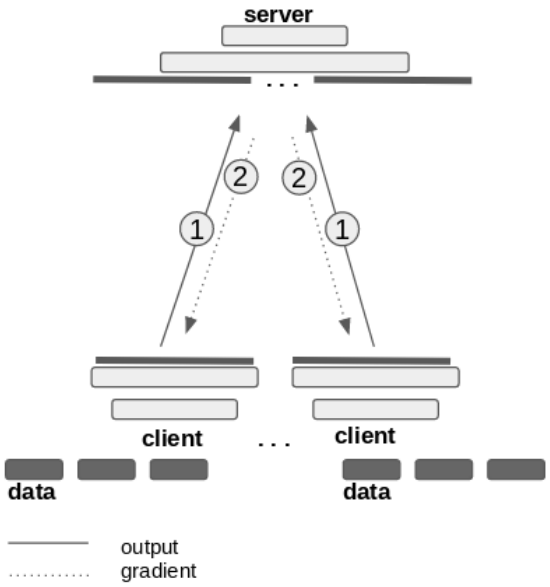

Figure 3.1: ParallelSplit Architecture

3.2 Federated Split Learning

Federated Split Learning, is a merge between the Split Learning [5] and the Federated Learning [1] algorithm. In fact, at each epoch, we compute the average weights of some layers of the neural network. In addition, each Federated Learning client is divided in two parts, $Alice_i$ and Bob_i, where range of i is between 1 and $|I|$.
This fact means that each client may not have a high computational power which is instead necessary if we use Federated Learning.
Probably a negative aspect could be that we need a number of servers equal to the number of clients. In any case, in some application, it could not be a big problem if we think, for example, that we can use some clouds. This architecture was composed by a number of clients that is equal to the number of servers. The forward propagation is computed by each client and the result of this computation is sent to the corresponding server, that finishes the forward propagation and computes the loss and the gradient. Then each server back propagates the gradient along the network. Finally, the back propagation continues along the client neural network.
When all the data of each client have been used to update the parameters, the weights of the servers' parts of the neural network are averaged and updated. In this way we are able to merge the information coming from the different clients using the same approach exploited by the *Federated Averaging Learning* [1].

Algorithm 5 FederatedSplit

initialize:

$\phi \leftarrow$ Random Initializer (e.g. Gaussian)

$F_{a,i} \leftarrow L_0, L_1, \ldots L_n \quad \forall \quad i \quad in \quad I$

$F_{b,i} \leftarrow L_{n+1}, L_{n+2}, \ldots L_N$

$Alice_i$ randomly initializes the weights of $F_{a,i}$ using $\phi \; \forall \; i$ in I

Bob_i randomly initializes the weights of $F_{b,i}$ using $\phi \; \forall \; i$ in I

for *epoch* in *epochs* **do**:

while $Alice_i$ has new data to train on **do**

X $\leftarrow F_{a,i}(data)$

Send((X, labels), Bob_i)

$output \leftarrow F_{b,i}(X)$

$gradient \leftarrow G'(output, labels)$

$F'_{b,i}$, gradient' $\leftarrow F^T_{b,i}$(gradient)

Send($gradient'$, $Alice_i$)

$F'_{a,i} \leftarrow F^T_{a,i}(gradient')$

end while

average of *weights* for each layer of the Bob model

end for

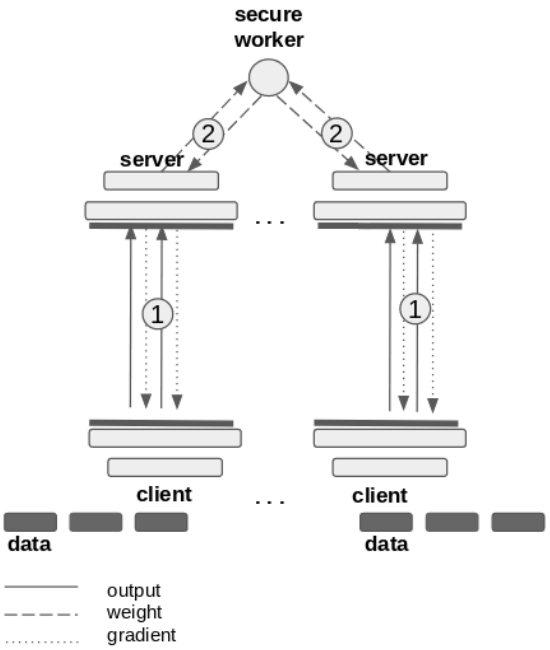

Figure 3.2: Federated Split Architecture

3.3 How to merge Clients' information

One of the most important differences between these two methods is how the two architectures merge the information from different clients. In this section, we detail how the merge is computed.

With Federated learning [1] and Split Learning [5] we obtain a global model at the end of the training phase. Using these two architectures, we merge the information of the last part of the CNN (after the *cut layer*) because they own the most specific part of the CNN.
We consider the first part of our CNN a way to extract the easier features, making input data safer than before and reducing the amount of information to send on the network. Usually, the first layers of a CNN act as an encoder able to find the straightforward features and, at the same time, summarize them in smaller images. Since the clients' data are different from each other, it is not necessary that each client has the same weights and uses the same first part of CNN. Therefore, we want to analyze how to make a summary of the second part of the CNN.

ParallelSplit can compute the merge as follows: the batch size of the server is equal to the sum of each batch size of all the clients. In this way, the loss function computed at each round is influenced by the first part of the CNN that is owned by each client.

$$\mathcal{B}_{server} = \sum_{i=0}^{N} \mathcal{B}_{client_i}$$

where $\mathcal{B}$ = number of images in batch size.

On the other hand, to compute the merge of all the information coming from the clients' data, the *FederatedSplit* calculates the average of all the weights of the server's part of the neural network.

Chapter 4

Performance Analysis

In this chapter we describe the analysis performed on the two previous architectures. Using the *PySyft* library [6] and we begin from the implementation on the Single Agent Split Learning moving to the Multi Agent Sequantial Split Learning and finally concluding with the two architectures.

4.1 Pysyft Library

The tool used in this work is the open-source library PySyft [6]. It is developed to make deep learning algorithms more private than before. In fact, if these kind of procedures exploit users' data to make a prediction or to solve a problem they should preserve the user's data as much as possible.
PySyft [6] is an extension of PyTorch, TensorFlow, and Keras. Basically, it is helpful to simulate and then create a remote execution for the distributed learning algorithm described in the previous chapter.
This library was divided into three main sections:

- Encrypted Computation: in order to send the data or other components of a deep learning model PySyft [6] develops a section based on cryptographic techniques: Multi-Party computation [13] and Homomorphic Encryption [14]. The first method is able to preserve privacy if a model is shared with more than one client. In general, a way to do this is the following: each different client owns a small part of the model and anyone can build the total model. The second one is used when all the model is owned by a single user. It tries to preserve privacy inside a machine in order to avoid that the model is stolen.

- Differential Privacy [15]: with really huge data it is possible to share data without getting private information from an individual, using the output of the final model trained on the dataset. In PySyft framework [6], there are

two types of differential privacy: general techniques, such as Laplace and Exponential mechanisms of DP-SGD and automatic DP that help the user to find the more private and better privacy tool to perform differential privacy for a certain goal and set of data.

- Remote Execution: It is another way to preserve privacy information. Instead of sending user's data to a server to train an algorithm it sends the model to each client in order to avoid sharing of private data and at the same time train the algorithm.
  This is the general idea behind Federated Learning. Furthermore, it can be used also to perform a prediction directly on a certain device without sending data on the network.

PySyft Workers

In order to achieve the result above we override all tensors' methods on the PyTorch. Working in this way PySyft [6] is able to extend the functions available to adapt them to another goal. In this section we describe the two different workers

Virtual Workers

In PySyft [6] there are a method to simulate the remote execution of the code: Virtual Workers. They are tensor pointer which are able to simultate remote machines.
Each workers are identified by an *Id* and authomatic added to a list of tensor pointers.
Two of the most important functions of these objects are: *send()*, *get()* and *move()*. The first one can send something that is inside a certain worker to another one passing through a local secure worker. This function is really useful to send the weights of a model to a worker for example during a federated learning algorithm. The second one is used to receive something that is owned by a Virtual Worker in local, into a safe machine. We used, for example this method to receive the final loss function obtained after the training phase or the value of the accuracy score. Finallly, the function *move()* is able to send directly a tensor from a Virtual Worker to another without using the local machine.
It is possible to see an example of these three methods in appendix A.

Remote Workers

Remote Workers are really similar to a Virtual Worker and they are called *WebsocketClientWorker*. In order to create one of this worker we should create a tensor pointer that contains all the information necessary to use a socket connection. In fact, we should specify an *Id* and the *Port* that represents the address to a remote machine.

4.2 Experiments on MNIST dataset

Dataset

The dataset that we used is MNIST [16]. MNIST is a dataset composed by 60000 training images and 10000 testing images of digits from 0 to 9. It was created from *Yann LeCun, Corinna Cortes and Christopher J.C. Burges* in 2010 as a subset of a larger NIST dataset.
The dimension of these images is 28x28 pixels.

Neural Network

For the MNIST dataset we used LeNet [17]. It is a convolutional neural network designed by *Yann LeCun* in 1998. It is one of the first Deep Neural Network and it became popular when it was used to recognize handwritten zip code digits in U.S. In the following picture 4.1 we presents a schema of this architecture. We can notice that each convolutional layer is followed by a non linear activation function (Tanh) and by an Average Pooling that can reduce the shape of the original image. A summary of this architecture is shown in Figure 4.2, in which we can see also the total amount and size of parameters.

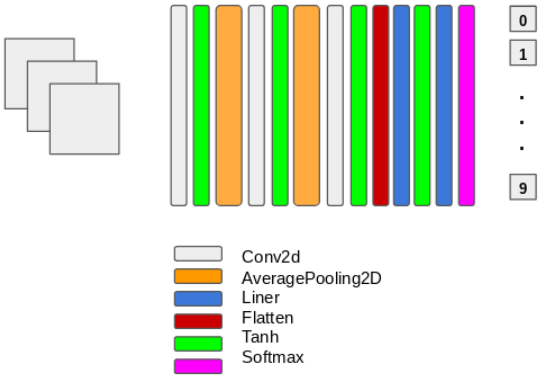

Figure 4.1: Lenet CNN

Single Split Learning

In this section we describe how to obtain the Single Split Learning using *PySyft* library [6].
In the beginning, we create a Simulation of the remote workers using the *VirtualWorkers* API. We then develop a class in which we overwrite the forward and the backward methods to send the activation function from the client to the server machine and send back the gradient in the opposite direction. In the schema 4.3, we present in detail how the Forward and Backward function work.

Furthermore, we can notice that the performance are the same respect to train

```
----------------------------------------------------------------------------------------------------
Layer (type:depth-idx)                   Output Shape              Param #
====================================================================================================
├─Conv2d: 1-1                            [-1, 6, 28, 28]           156
├─Tanh: 1-2                              [-1, 6, 28, 28]           --
├─AvgPool2d: 1-3                         [-1, 6, 14, 14]           --
├─Conv2d: 1-4                            [-1, 16, 10, 10]          2,416
├─AvgPool2d: 1-5                         [-1, 16, 5, 5]            --
├─Conv2d: 1-6                            [-1, 120, 1, 1]           48,120
├─Tanh: 1-7                              [-1, 120, 1, 1]           --
├─Flatten: 1-8                           [-1, 120]                 --
├─Linear: 1-9                            [-1, 84]                  10,164
├─Tanh: 1-10                             [-1, 84]                  --
├─Linear: 1-11                           [-1, 10]                  850
├─Softmax: 1-12                          [-1, 10]                  --
====================================================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
Total mult-adds (M): 0.42
----------------------------------------------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.05
Params size (MB): 0.24
Estimated Total Size (MB): 0.29
----------------------------------------------------------------------------------------------------
```

Figure 4.2: Summary Layers LeNet CNN

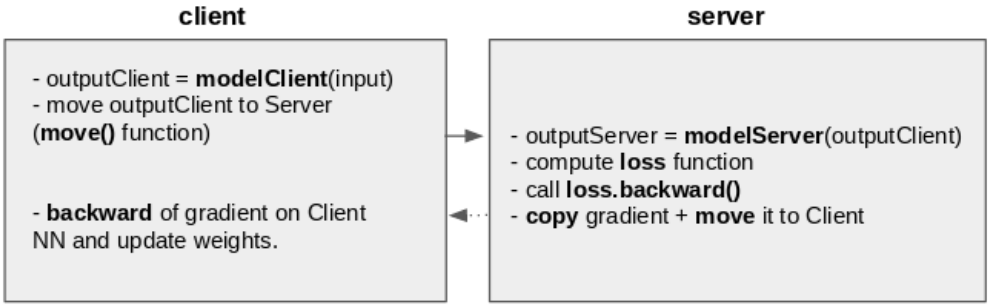

Figure 4.3: Single Split Learning Algorithm

all the model on the same machine because we divide the computational graph built by *PyTorch* in two parts and then we join it again.

Dynamic Computational Graph

To better understand how to divide the neural network workflow, we should focus on the comprehension of the process that *PyTorch* do every time an operation is computed.

PyTorch computes a graph called *Dynamic Computational Graph*, that is composed of a forward and a backward process.

To easily explain this approach, we show an example, and then we try to show how to split the two operations described below into two different VirtualWorkers. Finally, instead of simple operations, we used this approach with the two neural network parts. We can imagine computing these two operations:

$$C = A * B$$

$$E = C + D$$

where A,B,C,D,E are *torch.tensor()*. In particular we set:

- $A = torch.tensor(3.0)$
- $B = torch.tensor(4.0)$
- $D = torch.tensor(2.0)$

Sum and Product are also fundamental operations for a neural network.
As notable in the Figure 4.4, each *torch.tensor* is characterised by the following attributes:

- $data$: the value and the type of the variable
- $grad$: gradient (at the beginning set to $None$)
- $grad\_fn$: computational operation
- is_leaf : if the variable does not depend from any other operation (in our case A, B, D)
- $requires\_grad$: $True$ if we want to compute the gradient during the Backward propagation

Forward propagation
First, we compute the product between tensors A and B. To do this, we can see that the $grad\_fn$ of the result C has as value the word *MulBackward*; a context variable called *ctx* is used to save the value of the tensor and create a vector of two functions used during the Backward phase (in the figure: *[(AccumulateGrad, 0), (AccumulateGrad, 0)]*).
Then we compute the sum between the variable C and the variable D. This time the box created is called AddBackward.

Backward propagation
In order to get the gradient of each operation (it is, for example, essential inside a neural network) we called the function:

$$E.backward()$$

It computes the value of the gradient of the tensors used to compute E. Then if the tensor is a leaf, it calls the function *AccumulateGrad* that accumulates the gradients computed until that time and memorizes the result inside the attribute *grad* of the *leaf tensor*. The accumulation is made using the *Chain Rule*:

$$\frac{\partial z}{\partial x}\Big|_x = \frac{\partial z}{\partial y}\Big|_{y(x)} . \frac{\partial y}{\partial x}\Big|_x$$

where the function $z = y(x)$.
If the tensor is not a *leaf*, we send the gradient computed to the variable represented the previous operation (in the case in the example, we send the value 1.0 to the green box represented the *Mulbackward* operation).
Finally, in this case, we compute the next two gradients of the inputs data, and we accumulate the gradient. Furthermore, the tensors A and B are two *leaves* and so we can save the result in the attribute *grad*, and the backward propagation is ended.

Now, we can imagine splitting these two operations between two VirtualWorkers.

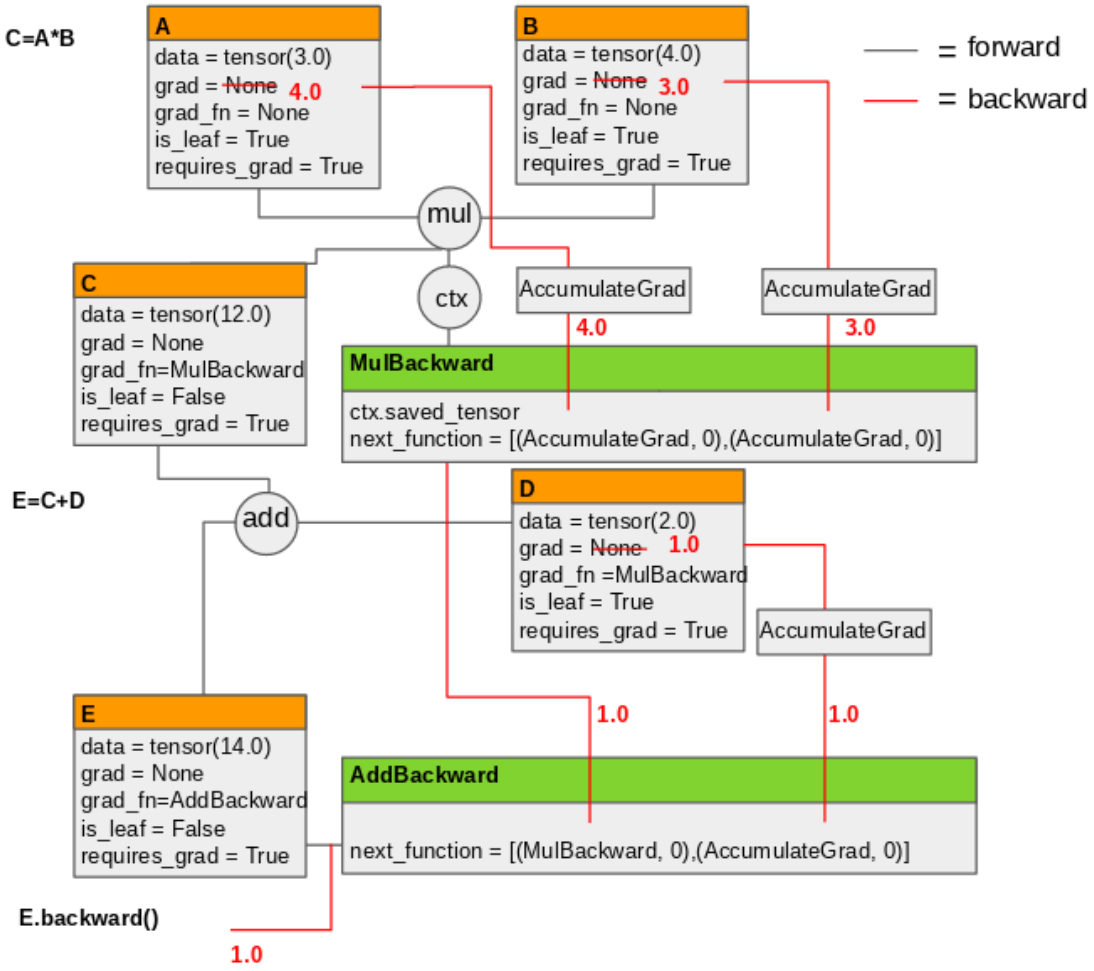

Figure 4.4: Dynamic Computational Graph

As we can expect by looking at the computational graph, we should find a way to cut and then rejoin the graph created inside *PyTorch*.

For the previous easy example, the operations that we should do are the following:

```
import torch

# Define the tensor A and B
A = torch.tensor(3.0)
B = torch.tensor(4.0)

# Set the attribute requires_grad equals to True to compute the
    gradient
A.requires_grad=True
B.requires_grad=True

# Compute product between A and B
C = A*B
```

To divide the computational graph we should detach the variable C computed in order to send it to the other machine (for example from the client to the server). To do this we use the function *detach()* and then the function *requires_grad* = *True*.

```
# Define variable I as:
I = C.detach().requires_grad_()

# Move the value I to the other machine (for example: client ->
    server)

# Define the tensor D
D = torch.tensor(2.0)

# Set the requires_grad equals to True
D.requires_grad=True

# Compute the sum between the I and D
E = I + D

# Call the backward function on E
E.backward()
```

We divided the computational graph, so, we can backpropagate and compute the gradient just until the tensor I. Therefore, copy the value of the attribute *grad* in the variable *grad_in* and we move back the value of the gradient from the server to the client (in this example).
We call the backward function on the vector C this time using as argument the value of grad moved back.

```
# Copy the value of the grad computed inside I
grad_in = I.grad

# Move back the gradient to the client

# Call the backward function on the tensor C
C.backward(grad_in)
```

Sequential Split Learning

After this approach, we implement the *Multi-Agent Split Learning* (*Sequential Split Learning* [5]) and we compare the performance respect the model obtained sending all the data on the same machine.
In Figure 4.5 we can notice that the value of accuracy obtained during the training and the validation phase in the local learning is more or less equal to the one got using 2 or more clients and the *Sequential Split Learning* as architecture.

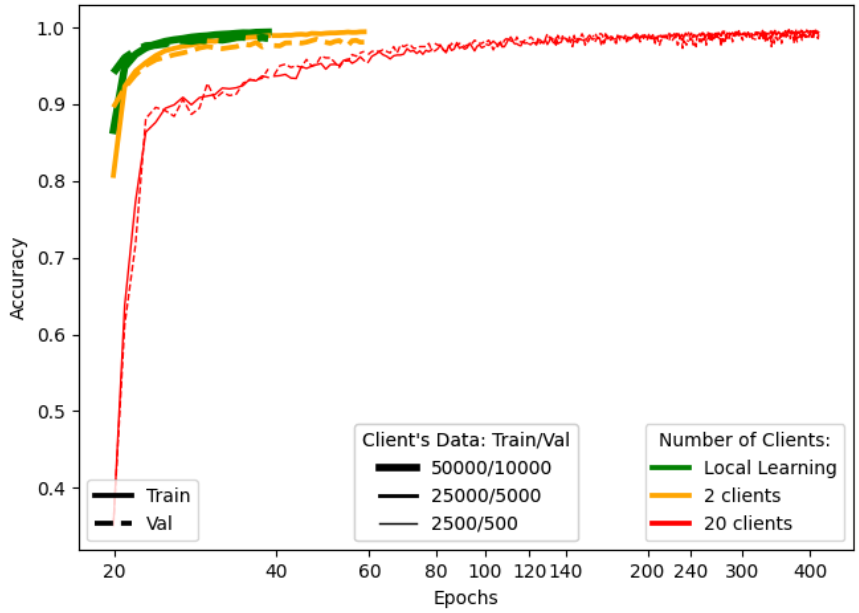

Figure 4.5: Accuracy Train/Val Set: Local Learning compared with Sequential Learning

Weaknesses

One of the most critical weaknesses of this method is that two or more clients cannot collaborate with the server at the same time.

We should wait that, for example, the first client has finished training the model before starting the training phase with the second one.
Furthermore, this method needs a trained client to send his weights to the following client before starting the training phase. The principle behind this approach is similar to transfer learning.
Therefore, a positive aspect can be that if the second client's dataset, in this example, is similar to the first one, this can speed the convergence of the second training phase. However, if it is different, the model might not even reach the convergence.
We will analyze this aspect in the following section.

Sequential and Parallel Split Learning

This section describes an experiment made on an unbalanced dataset to show in detail the most prominent drawbacks of the *Sequential Learning Algorithm*.
Dataset
The dataset used for this experiment is the MNIST dataset [16]. The most important difference between the previous approach is that we decide to use two clients, and we distributed the digits in the following way: we send the even digits to the first client and odd digits to the second one. We define with the parameter λ the percentage of data unbalance on each client.
As it is possible to observe in Figure 4.6, when the parameter λ goes toward the number 1, the first client has a large number of "even numbers," and the performance reached by the *Sequential Split Learning* are poor. If the clients' distribution is different, and their data distribution is unbalanced, this architecture may not reach convergence.
Algorithm 6 describes how to split the dataset between clients in an unbalanced way.

Algorithm 6 Client 1 - Dataset (majority of even digits)

```
initialize:
mask = [1 if trainset[i][1]%2 == 0 else 0 for i in range(len(trainset))]
  for i in range(len(mask)) do:
     if mask[i] == 0 then:
        if random.uniform(0,1)> λ then:
           mask[i] = 1
        end if
     end if
  end for
```

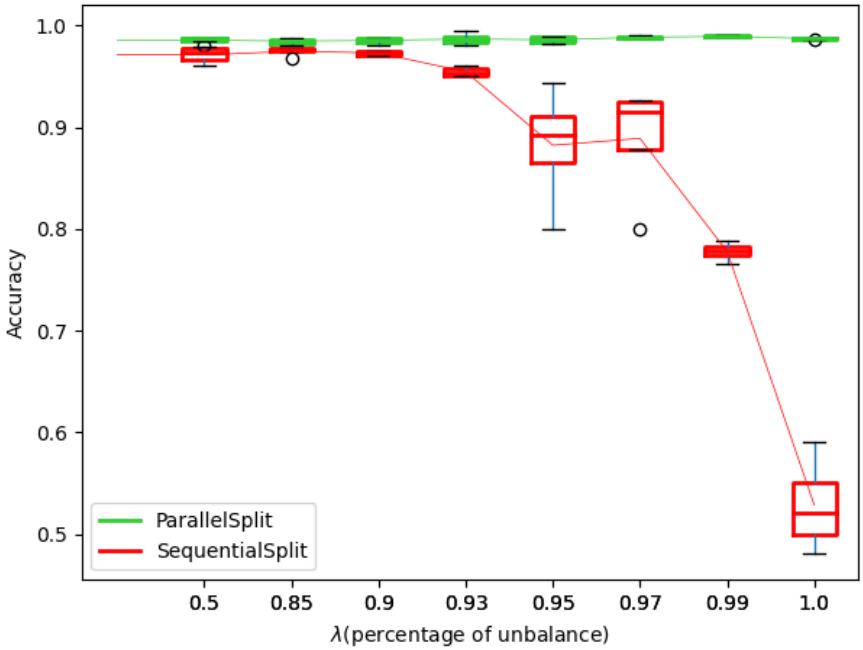

Figure 4.6: Comparison between SequentualSplit and ParallelSplit

Parallel Split and Federated Split Learning

In this section, we show the comparison between *Parallel Split* and *Federated Split Learning*. These two algorithms differ from how they compute the merge of the information between clients. In fact, in the first procedure, we use just one server, and the loss function computed by this machine can sum up all the results obtained by the different inputs arrived from each client. The second one is similar to the *Federated Averaging Learning* and computes the average of the weights computed by each server.

We show below two plots (Figure 4.7) related to the Loss function and the accuracy of the test set obtained using the MNIST dataset, and the LeNet neural network split at the third layer. To obtain these two plots, we divide the whole dataset between two clients in a balanced way.
As we can see, the accuracy obtained with the two architectures is similar and similar to the accuracy level reached on the local machine with the *Sequential Split Learning*.

Note how the accuracy changes if we change the number of data items that each client uses at the beginning of the training process. One of the federated learning

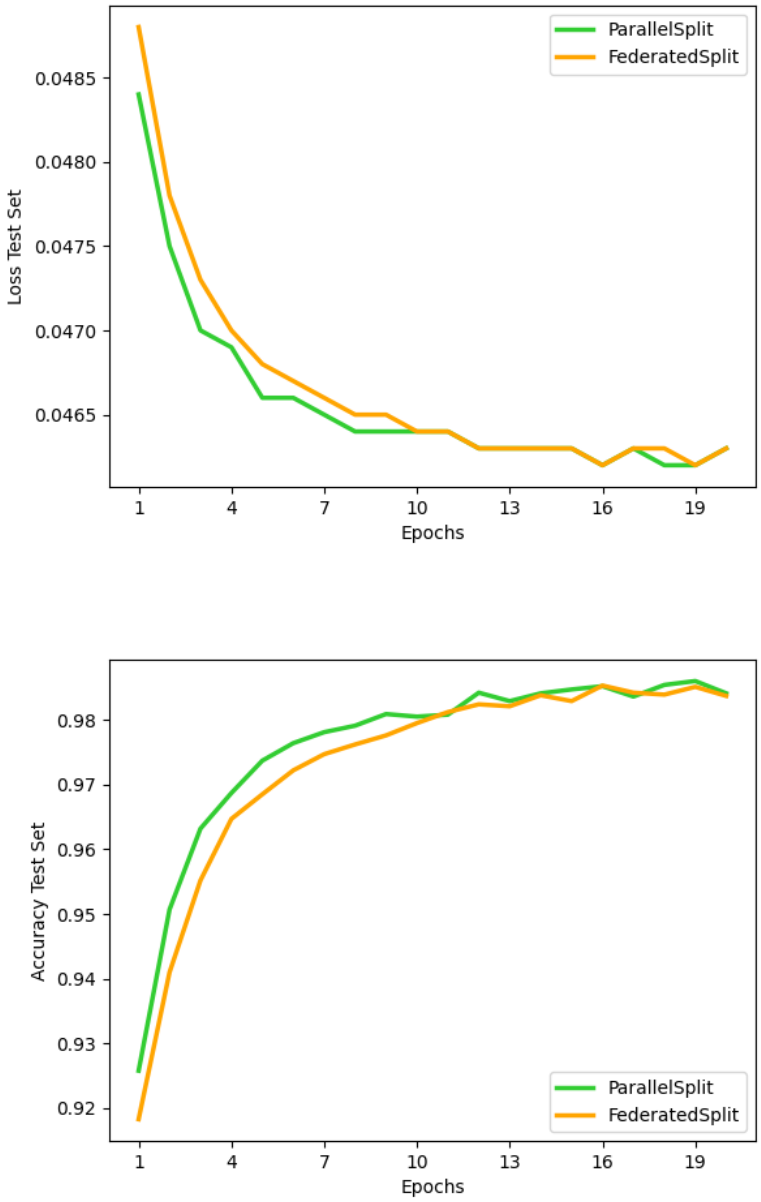

Figure 4.7: Top: Loss-Test, Bottom: Accuracy-Test for ParallelSplit and FederatedSplit

algorithms' issues is that if the dataset is massively distributed, the performance can degrade. The second architecture, *FederatedSplit*, is a mixture between the *Split Learning* and the *Federated Learning* approach; therefore, we can see in Figure

4.8 that it inherits the negative aspect owns by the Federated that if the clients have a small dataset, the performance reached by the neural network could be worse.
To do this experiment, we fix the dataset used (MNIST), and at the same, we try to increase the number of clients. Furthermore, the *cut level* was fixed to the third layer of the *LeNet* neural network.

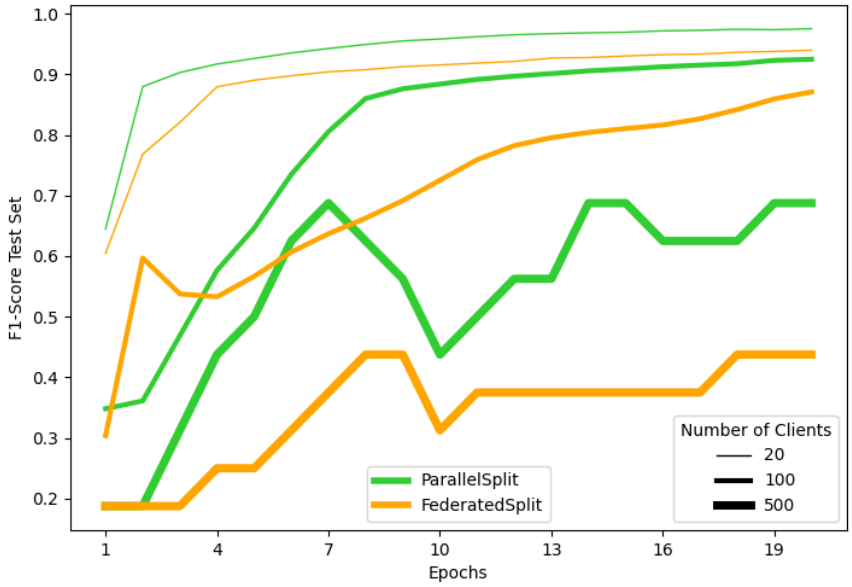

Figure 4.8: Compare ParallelSplit and FederatedSplit increasing the number of clients

Data to send on the network

In this section, we would like to estimate how many data we should send on the network for the two architectures that we described in the previous chapters.
In Table 4.3, we summarize the shape and the corresponding number of Megabytes for each batch of 32 images and for both the architectures; finally, we estimate the total number of Megabyte to send on the network for the training phase.
To compute these two estimations, we need to know the number of trainable parameters (in Megabyte) that the *LeNet* Neural Network uses to compute the classification. This is shown in the table 4.1.

Network Topology	Megabytes
LeNet	0.24
Server_LeNet	0.23
Client_LeNet	0.01

Table 4.1: Megabytes for LeNet parameters

Finally, we present in Table 4.3 the result of this estimation. To compute the total amount of Megabytes, we assume to compute the average of the weights for the *FederatedSplit Architecture* at the end of each epoch. We can notice that this can be reduced based on the result obtained by the model to avoid sending too much data on the network.
The column called *Tot(Mb)* represents the number of Megabytes that we should send on the network at each epoch.
It was computed using the following formula for *Parallel Split Learning*:

Tot (Mb) = ⌈ Training set / Batch size ⌉ x (Intermediate Result Batch + Gradient Batch)

and the following one for the *Federated Split Learning*:

Tot (Mb) = ⌈Training set/Batch size⌉ x (Intermediate Result Batch+Gradient Batch) + 2 x (Server LeNet parameters)

We can notice that the number 2 means that we send the servers' weights to a secure worker and then we send back the averaged weights from the secure worker to the servers.

Data	Shape	Megabytes	Training set (number of images)
Input	[32, 1, 32, 32]	0,131144	60000

Table 4.2: MNIST Batch Data

Dataset: MNIST				
Architecture	Name	Batch Shape	Megabytes	Tot(Mb)
ParallelSplit	Intermediate Result	[32, 6, 14, 14]	0,150600	$\sim$ 565
	Gradient	[32, 6, 14, 14]	0,150600	
SplitFederated	Intermediate Result	[32, 6, 14, 14]	0,150600	
	Gradient	[32, 6, 14, 14]	0,150600	$\sim$ 565
	Server_LeNet		0.23	

Table 4.3: Amount of data to send on the network at each epoch

Chapter 5

Privacy Attack and Solutions

One of the main problems of the Split Learning procedure is that the encoder part of the CNN, in other words, the part before the *cut layer*, could be reversed by an attacker who has enough data to train a decoder CNN.
In this chapter, we analyze some methodologies to avoid this fact to be sure that the privacy of the client's data is preserved, and then we show some results obtained using the *NoPeek* [12] approach.

5.1 Split-CNN attack

Using the Lenet CNN described above, we build a decoder CNN that has input images the intermediate output of the first layers of the CNN (encoder-CNN). As the previous experiments, we decided to cut CNN after the third layer.
The Encoder-CNN is composed by a Convolutional layer following by a Tangent Hyperbolic layer and finally by an Average Pooling that can reduce the dimension of the original image in order to summarize the most essential features of an image and, at the same time, reduce the amount of data to send on the network.

After this part, we built an attacker that is a Decoder-CNN. This part of CNN aims to invert the previous layers to define a measure of privacy. The more the reconstructed data differs from the initial one, the more the model can preserve data privacy. Using a large amount of data, we can reconstruct the input data using just this simple attacker. It has been implemented using two *Convolutional Transpose layers* that can reverse the early part of the CNN.

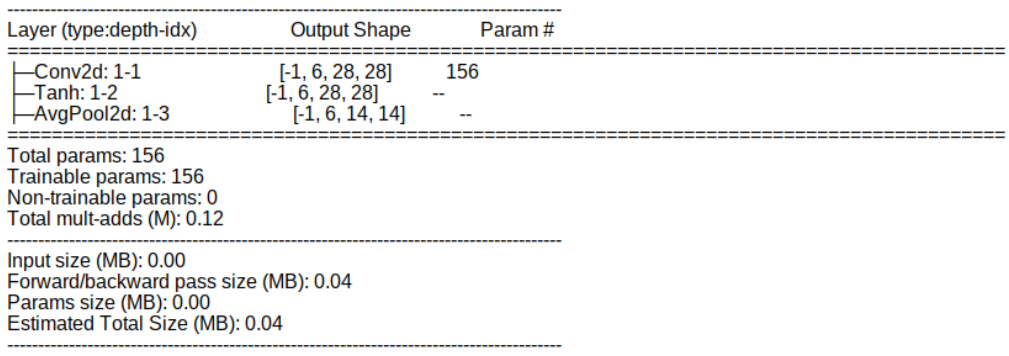

Figure 5.1: Encoder part of CNN

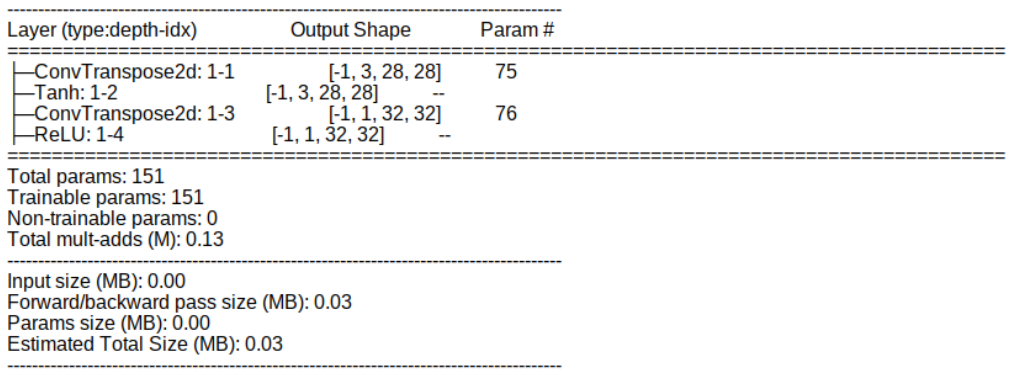

Figure 5.2: Decoder part of CNN

5.2 Privacy Techniques

In this section, we discuss some of the most famous techniques to avoid data loss during the training phase of a general distributed algorithm.

5.2.1 Homomorphic encryption and Secure MPC

These two techniques are the first two methods used to protect what we should send to a server in a distributed learning algorithm.
These two methodologies' main issues are that they are computationally expensive and not scalable to use during the training phase of a deep neural network.
In particular, the HE can compute operation on encrypted data without decrypting them [14]. At the end of the algorithm decrypting the output, we can obtain the same result that the algorithm without HE can produce. It was used in the federated Learning algorithm to protect the weights of the deep neural network that each client should send to a secure server to compute the average and merge clients' contribution.
Secure Multi-Party Computation [13] is another way to make the information

private. The idea is the following: we can imagine that a group of n clients should compute a function f, which has as input the client's input values.
The goal of this method is to preserve the privacy of those inputs. Furthermore, each client cannot obtain information from other clients.
These two methods are often used to compute simple functions, and they can be really slow and computationally expensive if they are employed in the computation of a deep neural network.

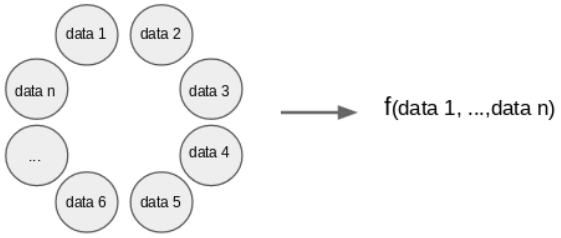

Figure 5.3: Secure Multi-Party schema

5.2.2 Differential Privacy

Differential privacy [15] can be used for huge dataset. It adds noise to data in order to preserve individual information. If the random substitutions of the original data are small enough, they cannot modify the algorithm's performance, but they can be able to preserve sensible information.

5.2.3 Increase Client's Neural Network

Another possible solution to deal with the privacy attack in *Split Learning* could be to increase the number of layers to give to each client. The Encoder's output is hence encrypted and, at the same time, more private than before. Furthermore, we should create an attacker with more layers and more weights to train, and consequently, more data are necessary.
We could try to find the correct balance between the number of layers to give to each client and the amount of memory and power required to train that part of the neural network. On the other hand, we should remember that we are under the assumption that the clients are devices with little computational power.

5.2.4 Reconstructive Adversarial Network

To find a way to overcome this kind of attack, the first approach that we want to study is described in [18].
The scheme below summarizes how the algorithm works. It optimizes at the same time two loss functions: the reconstruction error and the cross-entropy. The first one represents a measure of our algorithm; instead, the second one is a measure of the accuracy obtained.
The name of this attack is *RAN*: Reconstructive Adversarial Network, a deep

learning algorithm that can maximize at the same time privacy and utility.

It is based on a *max min* optimization. In other words, the loss function

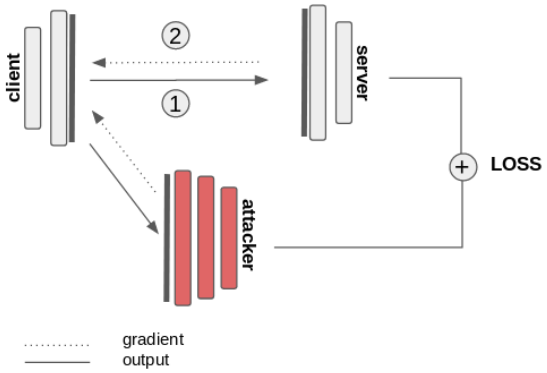

Figure 5.4: Adversarial Attack schema

that we want to optimize is the following:

$$\min_{E} \lambda \sum_{i=1}^{m} H_i - (1-\lambda) \sum_{i=1}^{m} MSE_i$$

where E is the Encoder Neural Network, m is the number of samples inside the batch size, H represents the cross-entropy and MSE is the mean square error that is the reconstruction error.
This function was obtained from the two following optimization problems:

$$\max_{E} Prob(Y_i' = Y_i)$$

that is the probability that the predicted label Y_i' is equal to Y_i and,

$$\max_{E} \min_{X} |I_i - I_i'|^2$$

that before minimize on the Decoder Neural Network to improve the reconstruction and then optimize on the Encoder Neural Network (client's part) to change the parameters to make the reconstruction as tricky as possible.
In other words, the network trains the Decoder (attacker's part) and the Classifier (server's part) in order to improve the reconstruction of the original image and, at the same time, obtain an adequate level of accuracy. However, it changes the weights of the Encoder neural network to avoid the reconstruction using the output of this part of the network.
We should correctly optimize the Encoder NN because otherwise, this can lead to

poor privacy with a possible reconstruction of the original data.
The algorithm 7 describes how to optimize the three parts of the neural network.

Algorithm 7 Algorithm to optimize Adversarial Neural Network [18]

```
for epoch in epochs do
    for on the data batch by batch do
        for k in K = number of local update do
            min H – update Classifier's and Encoder's weights
            min MSE – update Decider's weights
        end for
    end for
    min LOSS – sum of H and MSE update Classifier's and Encoder's weights
end for
```

The figure below represents a batch of 32 images; we can see the result of the reconstruction in 5.6: it is the output produced by the attacker's neural network. Note how the reconstructed images have low quality, and it is impossible to identify the original digits. Furthermore, this algorithm's accuracy remains more or less the same as the neural network trained without the attacker. One of the most

Figure 5.5: Original batch of 32 images

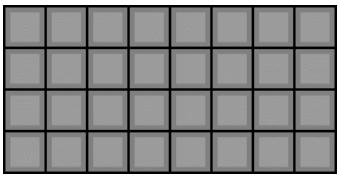

Figure 5.6: Reconstructed batch of 32 images

critical aspects of this method is that only at the end of the training phase, the Encoder can produce a safe image to send to the Classifier. We should send on the network all the training images until the convergence of the network is reached. Thus, it is not a good solution for our goal because we want that the images used during the training are private.

5.3 NoPeek Approach

The NoPeek algorithm is described in [12]. The authors tried to reduce the possibility to easily reconstruct the original image decoding the output of the activation function. This paper aims to find a way to improve the privacy level during a *Split Learning Algorithm* and, at the same time, maintain a high level of accuracy. The general idea consists of adding a part to the original loss function called *Distance Correlation*. Minimizing this measure, they try to minimize the dependence between the two images: the input and the output of the activation got from the client's neural network (the output of the *cut layer*).

5.3.1 NoPeek Theory

In the paper [19], the authors explain that one of the most important aspects of the distance correlation is that:

Theorem
For all the distribution X and Y with a finite mean: the distance correlation between X and Y is equal to zero if and only if X and Y are independent.
Furthermore, this measure can underline the linear and the non-linear relationship between the random variables.
In line with the standard *Person's Correlation* the authors of [19] define the *Distance Correlation* as:

$$dCor(X,Y) = \frac{dCov(X,Y)}{\sqrt{dVar(X)dVar(Y)}}$$

In particular, the *sample Distance Correlation* is derived from the definition of *Distance Covariance Correlation* and from *Distance Variance Correlation*:

$$dCov_n^2(X,Y) := \frac{1}{n^2}\sum_{j=1}^{n}\sum_{k=1}^{n} A_{j,k}B_{j,k}$$

$$dVar_n^2(X) := dCov_n^2(X,X) = \frac{1}{n^2}\sum_{j=1}^{n}\sum_{k=1}^{n} A_{k,j}^2$$

$$dVar_n^2(Y) := dCov_n^2(Y,Y) = \frac{1}{n^2}\sum_{j=1}^{n}\sum_{k=1}^{n} B_{k,j}^2$$

where, $A_{j,k}$ and $B_{j,k}$ are respectively:

$$A_{j,k} = a_{j,k} - \bar{a}_{j.} - \bar{a}_{k.} + \bar{a} \quad \forall k,j = 1,2,...,n$$

$$B_{j,k} = b_{j,k} - \bar{b}_{j.} - \bar{b}_{k.} + \bar{b} \quad \forall k,j = 1,2,...,n$$

where:

$$a_{j,k} = ||X_j - X_k|| \quad \forall k,j = 1,2,...,n$$

$$b_{j,k} = ||Y_j - Y_k|| \quad \forall k, j = 1, 2, ..., n$$

which the symbol ||.|| represents the Euclidean Norm, with the symbols $\bar{a}_{j,.}$ we represent the mean of the j-th row, $\bar{a}_{.,k}$ k-th column mean and finally with $\bar{a}$ the mean of of the vectors in the distance matrix. Therefore, they exploited this concept to provide a method that can, at the same time, maximize the accuracy of the model and data privacy.s They implement this method on the *Single Split Learning Architecture*. In order to do that the standard loss function (for example Cross Entropy) was changed in the following way [19]:

$$F(X, Z, Y, \bar{Y}) = \alpha_1 dCor(X, Z) + \alpha_2 CCE(Y, \bar{Y})$$

where α_1 and α_2 are two scalars, X is the input data, Z output of the activation function that results from the client's neural network, Y true label, and $\bar{Y}$ predicted label. The optimization problem behind this approach is trying to minimize the loss function described above using the gradient descent and forward/backward propagation.

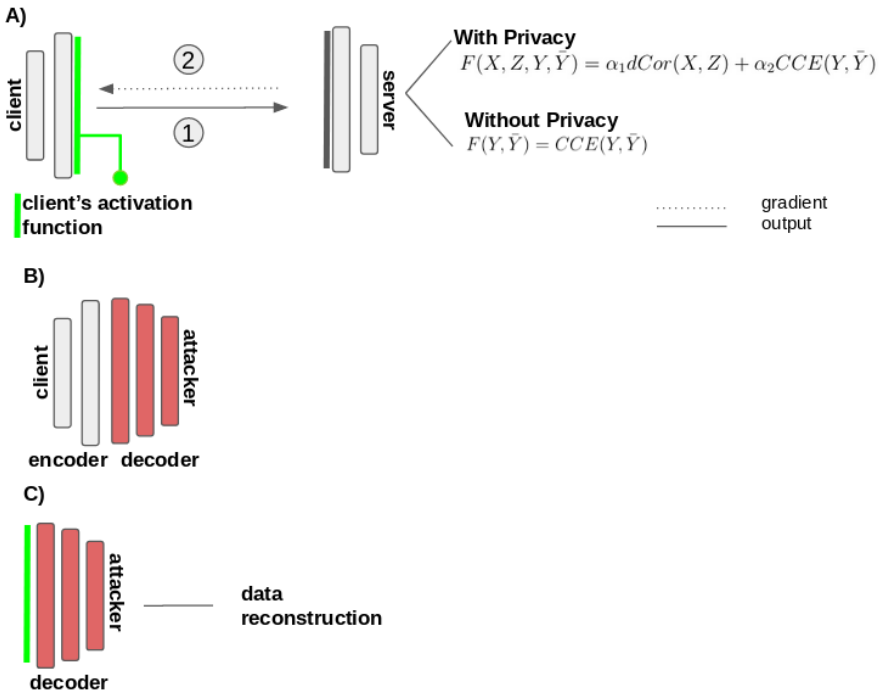

Figure 5.7: **A)** Split Learning Training using with/without Privacy on Original dataset; **B)** Train the Attacker Neural Network on Attacker's dataset; **C)** Data Reconstruction from the client's activation function (Decoder in inference phase)

5.3.2 Experiment

We decide to apply the *NoPeek* method to the two architectures described above: *ParallelSplit* and *FederatedSplit* to see if this method can improve the privacy of the data and avoid the reverse-attack for these two distributed methods.
Therefore, we replace the *Single Split Learning Algorithm* in Figure 5.7 **A)** with the two architectures.
To do this, we add the *Distance Correlation* to the *Cross Entropy*. The process is different for both architectures. How the two algorithms compute the loss function is not the same.
In particular, for the *ParallelSplit* we have just one server that can compute the loss function adding the *Distance Correlation* of each client.
Instead, for the *FederatedSplit* and for each pair of client and server, we compute a specific loss function.
Finally, the client's contribution is merged, computing the average of the servers' weights of the neural network.
In particular, the parameters used for the *ParallelSplit* and the *FederatedSplit* are the following:

Learning rate	$3e^{-}4$
Epochs for Distributed Training	20
Number of clients	2
Cut Layer	$3rd$
Initial Topology	LeNet

Table 5.1: Parameters used for *ParallelSplit-NoPeek* Training

Learning rate	$1e^{-}4$
Epochs for Distributed Training	20
Number of clients	2
Cut Layer	$3rd$
Initial Topology	LeNet

Table 5.2: Parameters used for *FederatedSplit-NoPeek* Training

In order to simulate as much as possible the presence of an attacker, based on the GitHub repository [20], we use a subset of the EMNIST dataset [21] (in the Figure 5.7 we called this one: Attacker's dataset). It is composed of handwritten letters of the alphabet. We decide to use this one instead of the MNIST dataset with digits to train the attacker to reverse the clients' neural network because we try to emulate the dataset that an attacker could have to train his part of the neural

network.
The decoder part of this neural network is the same used above in section *RAN*, and the encoder is represented by the client's neural network (in other words, the part of the global convolutional neural network before the *cut layer*).
In the images below, we show the result obtained after the training of the attacker-Neural Network. We try to train the network using a different amount of data; we use a subset of 500 and 5000 images of the EMIST dataset.
The left column of the picture 5.8 and 5.9 represent the original images of a batch; instead, the right column is composed of the reconstructed images after training the *Attacker's Neural Network* for 20 epochs.
We can see that with just 500 images, the reconstruction is not accurate. Instead, increasing the number of images using for the train the *Attacker's Neural Network* seems more or less identical to the original.

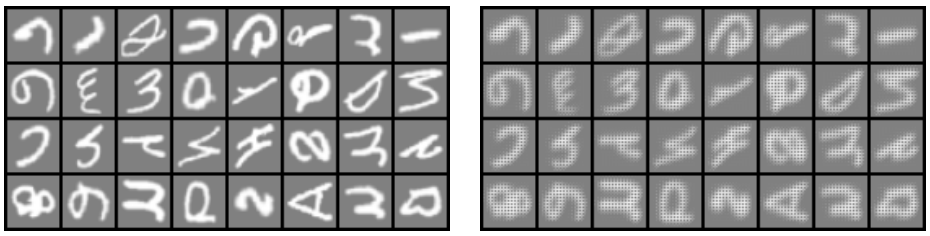

Figure 5.8: Subset of 500 images

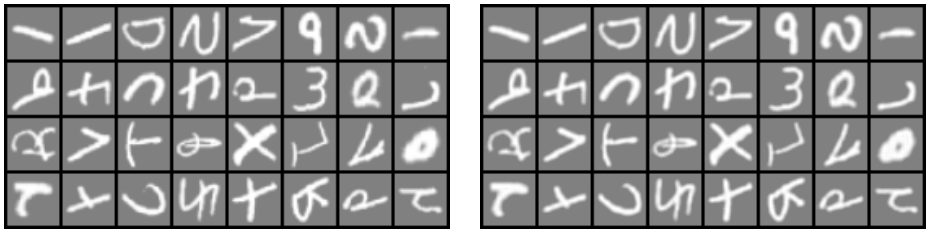

Figure 5.9: Subset of 5000 images

Figure 5.10: MNIST - Left: Original Batch of images; Right: reconstructed Batch of images using the Attacker's Neural Network

The Figures 5.14 and 5.18 show the result obtain in both the architectures using the NoPeek approach to avoid data reconstruction.
As it is possible to see, both algorithms can preserve the privacy of data. The left column represents the data reconstruction without the *NoPeek* approach instead of the right column the data reconstruction with the *NoPeek* approach.

Figure 5.11: Input Batch of images

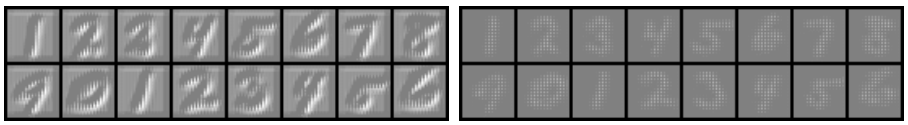

Figure 5.12: Subset of 500 images

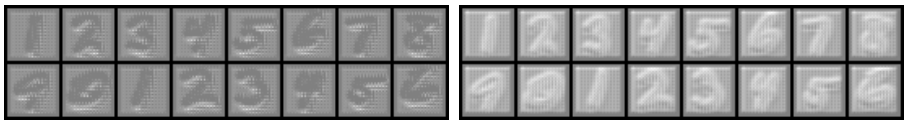

Figure 5.13: Subset of 5000 images

Figure 5.14: ParallelSplit - Left: Reconstruction without Privacy, Right: Reconstruction with Privacy. The rows represent the result of the input images get using the Attacker Neural Network trained respectively with 500, 5000 images.

Figure 5.15: Input Batch of images

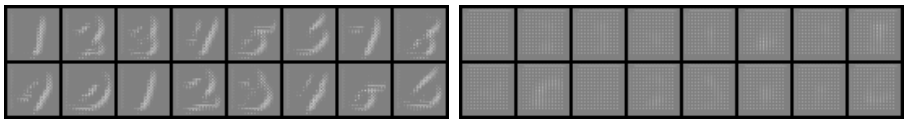

Figure 5.16: Subset of 500 images

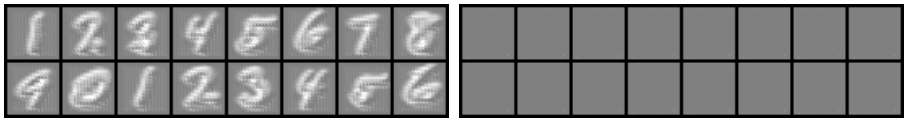

Figure 5.17: Subset of 5000 images

Figure 5.18: FederatedSplit - Left: Reconstruction without Privacy, Right: Reconstruction with Privacy. The rows represent the result of the input images using the Attacker Neural Network trained respectively with 500, 5000 images.

Table 5.3 shows the value of the Distance Correlation computed between the intermediate result and the input images for both the architectures with and

without the privacy approach.

Distance Correlation	Without Privacy	With Privacy
ParallelSplit	0.9964	0.9944
FederatedSplit	0.9958	0.9085

Table 5.3: Distance Correlation

Furthermore, as is possible to notice in the following graph for both architectures, the level of loss reached with the *NoPeek* approach is higher than the one reached without privacy. This fact is in line with what we might expect because the term we added to the loss function can make the intermediate output as independent as possible from the input data.
Finally, as a consequence of the previous statement, we can see that the accuracy level for the model with privacy is lower than the result obtained from the model without *NoPeek*.

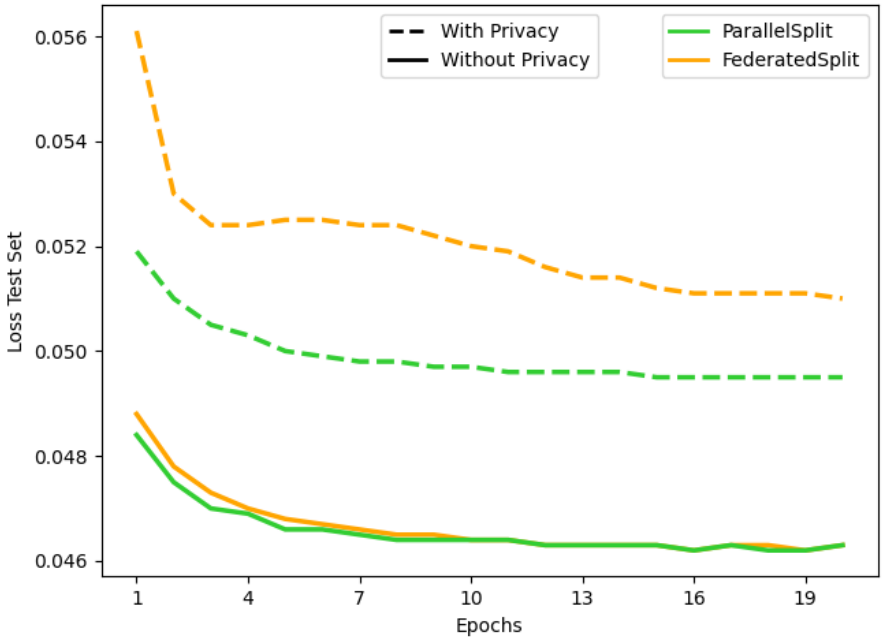

Figure 5.19: Loss -Test for ParallelSplit and FederatedSplit with and without Privacy

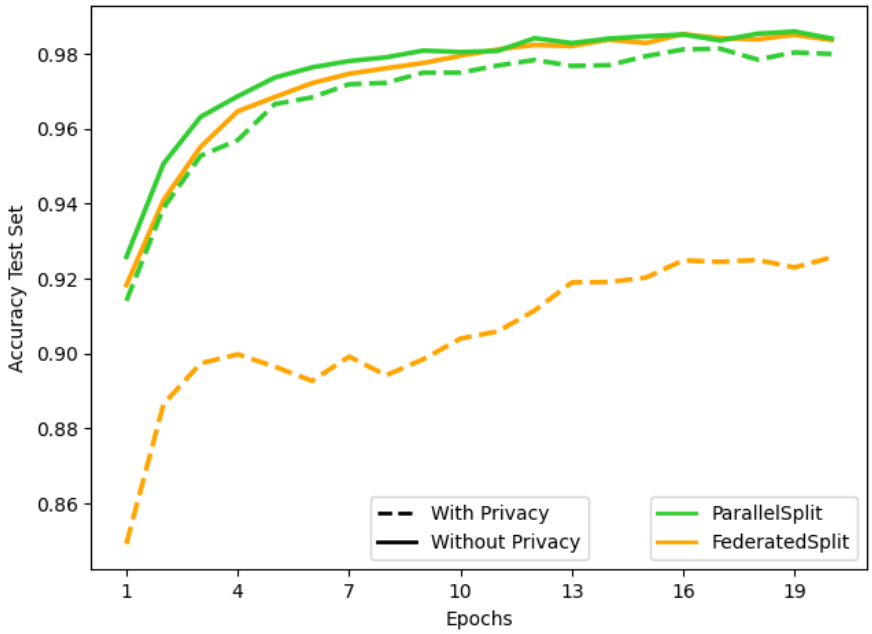

Figure 5.20: Accuracy-Test for ParallelSplit and FederateSplit with and without Privacy

Chapter 6

Health Application

In this chapter, we apply the previous results on a health application: the classification between COVID-19 and healthy chest using x-ray images.

6.1 Covid-19 Dataset

The COVID-19 dataset is composed of 625 CT images [22]: 125 related to COVID-19 affected patients and 500 of NO-COVID-19 patients; we can observe that even if this proportion does not correspond to the definition of an unbalanced dataset, there is a class (NO-COVID-19, also called No-Findings) that is more present.
Since we have a small dataset, we used a data augmentation approach in order to prevent over-fitting.
We used the *Pytorch Dataloader* function that has as parameter an object called $transform$. It give us the possibility to specify what kind of transformations the *Dataloader* will do on our dataset to augment the images.
First of all we resized the images to 256 x 256 pixels because the original data have different shapes. Then we decide to use:

- Random Horizontal Flip with a probability p = 0.5
- Color Jitter with brightness = 0.2, contrast = 0.2

Finally, we transformed the images to tensors to send them to a neural network. Using these transformations, the training set is randomly modified at each epoch. It must be noticed that this approach was performed only on the training set while concerning the test set and the validation set, we just resized and transformed the images to tensors.

6.2 DarkCovidNet

We built a CNN architecture for the binary classification task (NO-COVID-19 vs. COVID-19), starting from the DarkCovidNet in the reference paper [23].
The following graph illustrates the chosen architecture.

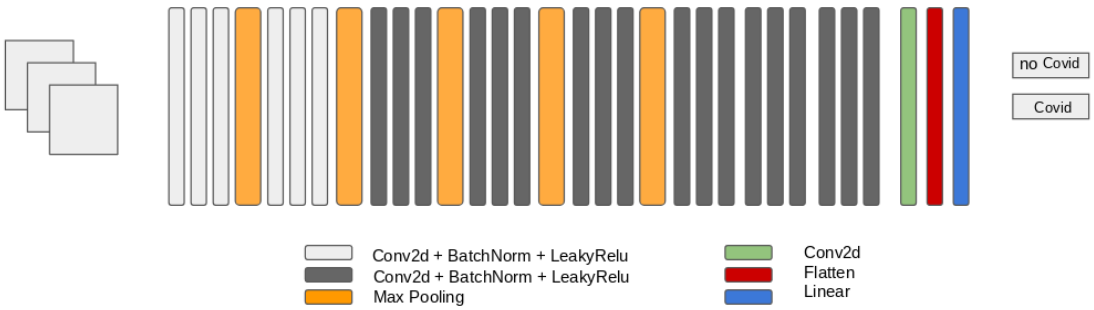

Figure 6.1: DarkCovidNet

To send the images to the model, we make them fit the first convolutional layer's input size, which is 256×256 pixels. Each image has three input channels for the CNN that correspond to RGB channels.
In our model, the convolutional layers had 3×3 kernel size, stride equal to 1, and zero-padding. Concerning max-pooling layers, the filter size is 2×2 pixels, with stride equal to 2.

6.3 Experiment

We try to reproduce the same experiments made with the MNIST dataset also with the COVID-19 CT images.
First of all, we decide to split the *DarkCovidNet* Model into two parts (the first one owned by a client and the second one by a server).
Looking to the schema of the model 6.1, we consider the second MaxPooling layer a good cut level for the model. Cutting the network after this layer means cutting it after six operations of convolution. This can improve privacy instead of choosing a previous layer. Simultaneously, after two operations of MaxPooling, the image of output is smaller than before.
If we increase the number of clients, fixing the batch size, we can see similar behavior compared to the MNIST Dataset. In fact, in Figure 6.2, we notice that distributing dataset on more than clients, the performance obtaining with *FederatedSplit* is worse than the one computed with *ParallelSplit*.

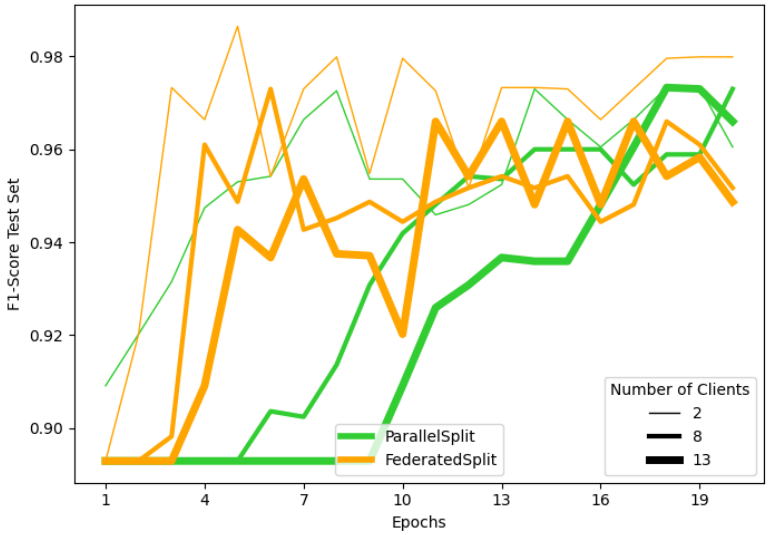

Figure 6.2: Compare ParallelSplit and FederatedSplit increasing the number of clients

Data to send on the network

In this section, we try to estimate the amount of data that we should send on the network using, like topology, the *DarkCovidNet*, and divide the network after the second *MaxPooling* layer. Table 6.1 represents the number of megabytes of the trainable parameters of the *DarkCovidNet* split between Client and Server, instead of the Table 6.2 represents the batch-Megabytes that we should send on the network in the case of a centralized training.

Network Topology	Megabytes
DarkCovidNet	4.44
Server_DarkCovidNet	4.43
Client_DarkCovidNet	0.01

Table 6.1: Megabytes for DarkCovidNet parameters

Data	Shape	Megabytes	Training set (number of images)
Input	[32, 1, 256, 256]	25,165896	500

Table 6.2: Batch Data

Finally the Table 6.3 summarizes the amount of data that we should send on the

network at each epochs using the *ParallelSplit* and *FederatedSplit* architecture. The formula computed to obtain the *Tot* amount of Megabytes is the same using for the MNIST Dataset.

Dataset: Covid				
Architecture	Name	Batch Shape	Batch(Mb)	Tot(Mb)
ParallelSplit	Intermediate Result	[32, 16, 64, 64]	8,38868	$\sim$ 268
	Gradient	[32, 16, 64, 64]	8,38868	
SplitFederated	Intermediate Result	[32, 16, 64, 64]	8,38868	
	Gradient	[32, 16, 64, 64]	8,38868	$\sim$ 277
	Server_DarkCovidNet		4.43	

Table 6.3: Amount of data to send on the network at each epoch

Privacy

In this section, we reproduce the privacy experiment described in Figure 5.7 using the two architectures instead of the *Single Split Learning*.
The topology used by the Attacker is described in Figure 6.3 and 6.4.

```
----------------------------------------------------------------------------------------------------
Layer (type:depth-idx)                   Output Shape              Param #
====================================================================================================
|   —Sequential: 2-1                     [-1, 8, 256, 256]         --
|   |   └─Conv2d: 3-1                    [-1, 8, 256, 256]         216
|   |   └─BatchNorm2d: 3-2               [-1, 8, 256, 256]         16
|   |   └─LeakyReLU: 3-3                 [-1, 8, 256, 256]         --
|   └─MaxPool2d: 2-2                     [-1, 8, 128, 128]         --
|   └─Sequential: 2-3                    [-1, 16, 128, 128]        --
|   |   └─Conv2d: 3-4                    [-1, 16, 128, 128]        1,152
|   |   └─BatchNorm2d: 3-5               [-1, 16, 128, 128]        32
|   |   └─LeakyReLU: 3-6                 [-1, 16, 128, 128]        --
|   └─MaxPool2d: 2-4                     [-1, 16, 64, 64]          --
====================================================================================================
Total params: 1,416
Trainable params: 1,416
Non-trainable params: 0
Total mult-adds (M): 33.03
----------------------------------------------------------------------------------------------------
Input size (MB): 0.75
Forward/backward pass size (MB): 12.00
Params size (MB): 0.01
Estimated Total Size (MB): 12.76
----------------------------------------------------------------------------------------------------
```

Figure 6.3: Encoder DarkCovidNet

```
----------------------------------------------------------------------------------------------------
Layer (type:depth-idx)                   Output Shape              Param #
====================================================================================================
|   —ConvTranspose2d: 2-1                [-1, 8, 128, 128]         520
|   └─Tanh: 2-2                          [-1, 8, 128, 128]         --
|   └─ConvTranspose2d: 2-3               [-1, 3, 256, 256]         99
|   └─ReLU: 2-4                          [-1, 3, 256, 256]         --
====================================================================================================
Total params: 619
Trainable params: 619
Non-trainable params: 0
Total mult-adds (M): 14.68
----------------------------------------------------------------------------------------------------
Input size (MB): 0.25
Forward/backward pass size (MB): 2.50
Params size (MB): 0.00
Estimated Total Size (MB): 2.75
----------------------------------------------------------------------------------------------------
```

Figure 6.4: Decoder DarkCovidNet

In order to train this network, the Attacker uses chest x-ray images with

Pneumonia or without pathologies [24].
Figures 6.5 and 6.6 show the result after 20 epochs of training. As it is possible to notice with 5000 images, the decoder can reconstruct the original images.

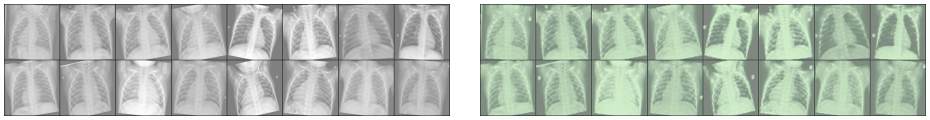

Figure 6.5: Subset of 500 images

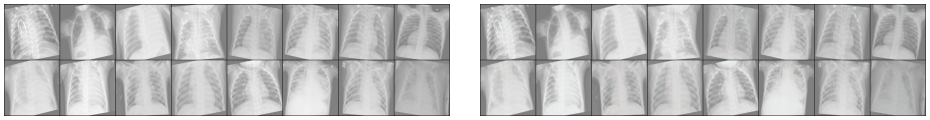

Figure 6.6: Subset of 5000 images

Figure 6.7: Left: Original Batch of images; Right: reconstructed Batch of images

In Table 6.4, we compute the Distance Correlation values between the intermediate results and the input images of the last batch after twenty epochs of training. As it is possible to notice the value of Distance Correlation obtaining using the privacy approach is lower in the *FederatedSplit Learning*. This means that the intermediate result is more independent from the input data than the one obtained using the *ParallelSplit Learning Algorithm*.

Distance Correlation	Without Privacy	With Privacy
ParallelSplit	0.9967	0.9848
FederatedSplit	0.9955	0.8865

Table 6.4: Distance Correlation

Finally, Figures 6.11 and 6.15 describe the result obtaining trying to reconstruct one of the intermediate results for both the architectures trained on the Dataset [22] using the *NoPeek* Approach.

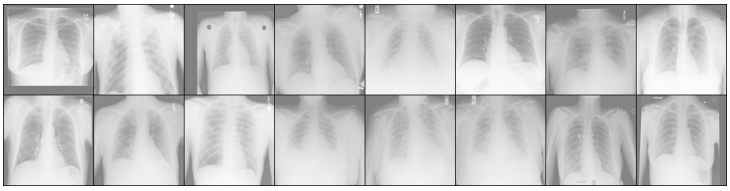

Figure 6.8: Input Batch of images

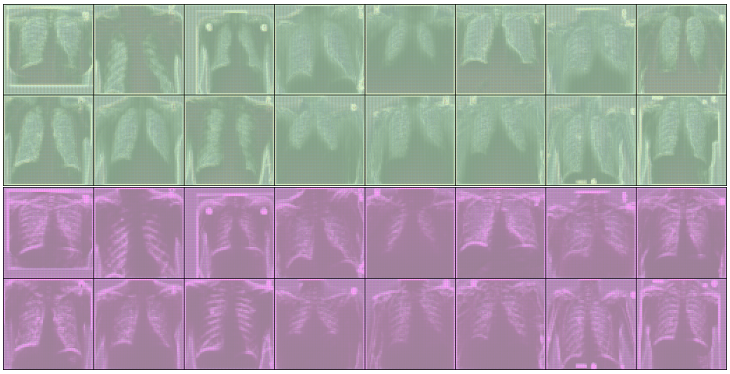

Figure 6.9: Subset of 500 images

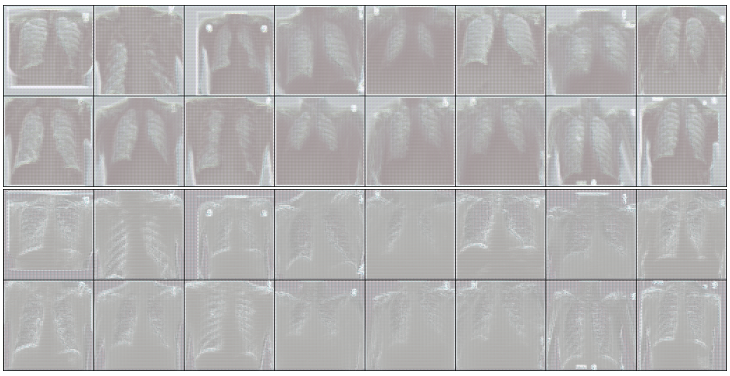

Figure 6.10: Subset of 5000 images

Figure 6.11: ParallelSplit - Top: Reconstruction without Privacy, Bottom: Reconstruction with Privacy. for both the results obtained the Attacker Neural Network trained respectively with 500, 5000 images.

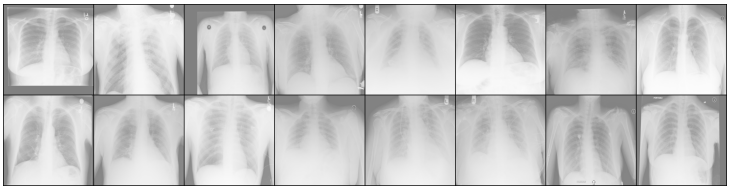

Figure 6.12: Input Batch of images

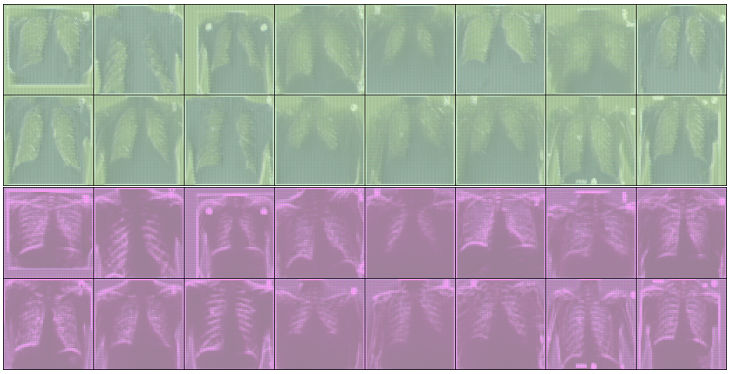

Figure 6.13: Subset of 500 images

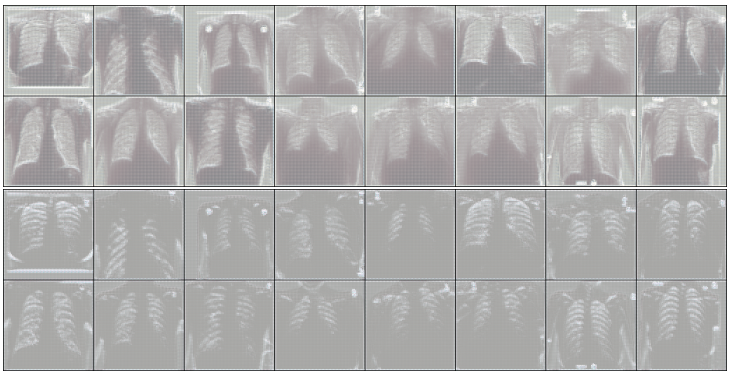

Figure 6.14: Subset of 5000 images

Figure 6.15: FederatedSplit - Top: Reconstruction without Privacy, Bottom: Reconstruction with Privacy. for both the results obtained using the Attacker Neural Network trained respectively with 500, 5000 images.

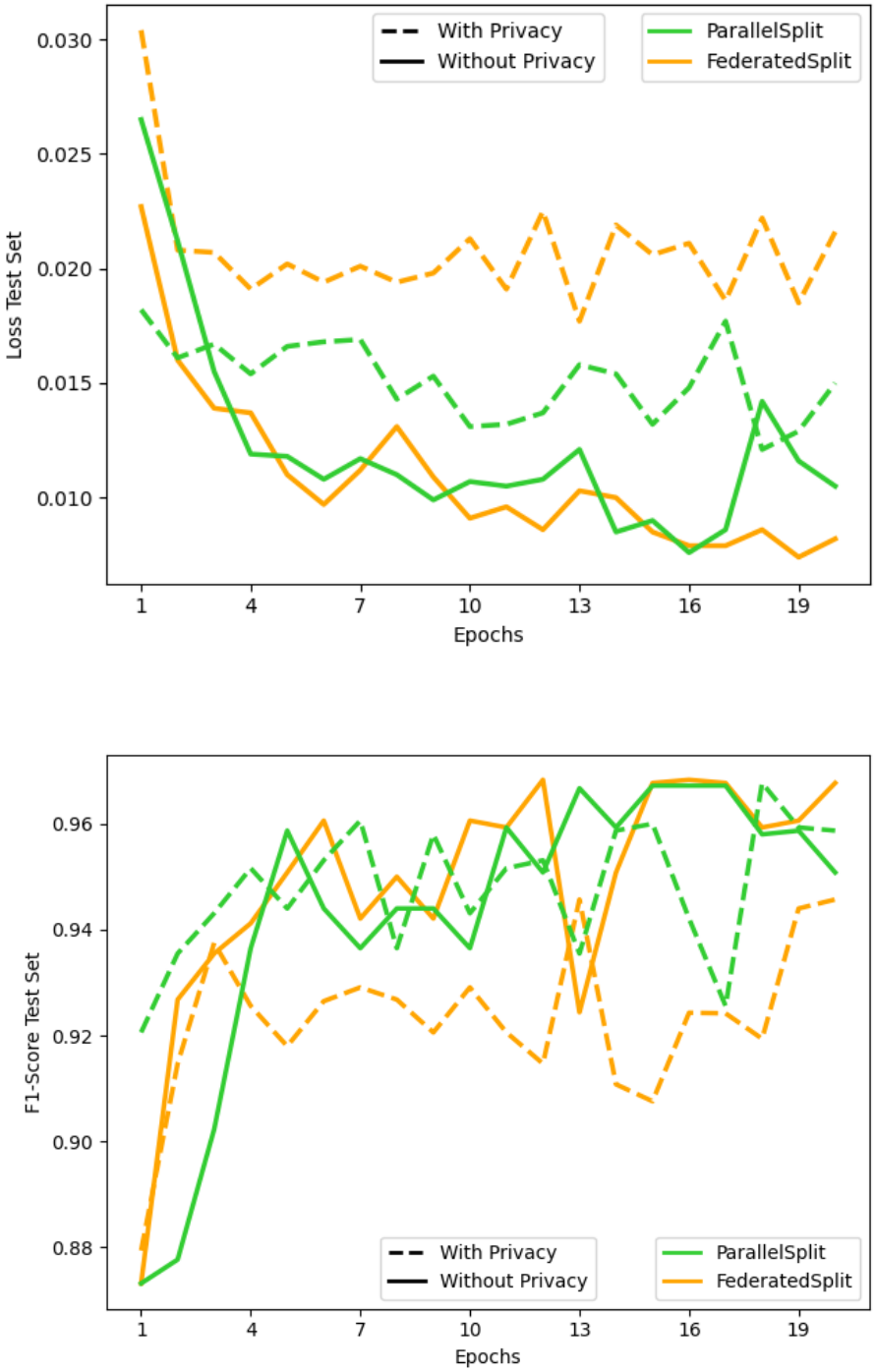

Figure 6.16: Top: Loss-Test Set for ParallelSplit and FederatedSplit with and without Privacy; Bottom: F1-Score-Test Set for ParallelSplit and FederatedSplit with and without Privacy

Chapter 7

Conclusion

In this thesis, we implemented two architectures that combine the Split and Federated learning algorithms using the *PySyft* library based on *PyTorch*.
The code was designed and implemented to easily switch from "local" to "remote" learning. We compared the two architectures in terms of efficiency and privacy, changing the data distribution between the clients, monitoring the accuracy level reached with respect to the number of iterations, and analyzing the privacy leakage.
We notice that if we reduce the quantity of data owned by each client, increasing the number of clients, the *FederatedSplit* architecture is worse than *ParallelSplit* in terms of accuracy reached. But at the same time, if we consider the level of privacy for the *FederatedSplit*, it is higher respect to the one obtained with the *ParallelSplit*.
Finally, we tested our algorithms on real-world datasets, such as MNIST and a medical dataset (COVID-19 CT chest images), where privacy is critical.

Appendix A

Appendix

A.1 VirtualWorker's functions

In this section we explain three functions of *PySyft* Library used in the code to send and receive tensors between different VirtualWorkers [25].

First of all after overwrite the hook method inside *PyTorch*;then to better explain these functions, we decide to declare two VirtualWorkers: *client* and *server*. They are two entities that can use as different machines inside the same laptop. They are really useful to implement an algorithm that in the future should be run on real different machine.
They are identified by an *id*

```
import syft as sy
import torch

hook = sy.TorchHook(torch)
client = sy.VirtualWorker(hook, id="client")
server = sy.VirtualWorker(hook, id="server")
```

The function *send()* can send the tensor from a local owner (for example a pc) to a certain VirtualWorker (in this example: id = "client"); but it can be used also to send a tensor from a VirtualWorker to another VirtualWorker (in this example: from id = "client" to id = "server").

Therefore, we define a *Torch* tensor called v and we show how to send it from the local machine (called: *me*) to respectively *client* and *server*.

```
v = torch.tensor([1,2,3])
v = v.send(client)
v = v.send(server)
```

```
# Output of v:
(Wrapper)>[PointerTensor | me:89712727978 -> client:39290507290]
(Wrapper)>[PointerTensor | me:98378852286 -> server:89207102262]
```

We can see that the *PointTensor* of the tensor v after these two operations points from the local machine to the *client* and from the local to the *server*.
The *client* entity has inside the tensor v:

```
# Client's Output:
{35637621589: tensor([1, 2, 3])}
```

Instead, the *server* entity has inside the following *Wrapper PointTensor*:

```
# Server's Output:
(Wrapper)>[PointerTensor | server:87666169779 -> client:35637621589]
```

Therefore, the tensor is on the *client* machine and there is a *PointTensor* of the tensor v on *server* that points to the *client* device.

In order to directly move the tensor we can use the function *move()*. After sending the tensor to the *client* we can move this tensor to the server using this function.

```
v = torch.tensor([1,2,3])
v = v.send(client)
v = v.move(server)
```

```
# Client's and Server's Output:
{}
{36989781708: tensor([1, 2, 3])}
```

Finally, we can receive the output that is owned by a VirtualWorker to the local machine using the function *get()*:

```
v = torch.tensor([1,2,3])
v = v.send(client)
v = v.get()
```

Bibliography

[1] H. McMahan, Eider Moore, D. Ramage, S. Hampson, and Blaise Agüera y Arcas. «Communication-Efficient Learning of Deep Networks from Decentralized Data». In: *AISTATS*. 2017 (cit. on pp. 11, 14–16, 26, 28).

[2] Andrew Hard, Chloé M Kiddon, Daniel Ramage, Francoise Beaufays, Hubert Eichner, Kanishka Rao, Rajiv Mathews, and Sean Augenstein. *Federated Learning for Mobile Keyboard Prediction*. 2018. URL: https://arxiv.org/abs/1811.03604 (cit. on p. 11).

[3] Francoise Beaufays, Kanishka Rao, Rajiv Mathews, and Swaroop Ramaswamy. *Federated Learning for Emoji Prediction in a Mobile Keyboard*. 2019. URL: https://arxiv.org/abs/1906.04329 (cit. on p. 11).

[4] Micah J Sheller, G Anthony Reina, Brandon Edwards, Jason Martin, Spyridon Bakas. «Multi-Institutional Deep Learning Modeling Without Sharing Patient Data: A Feasibility Study on Brain Tumor Segmentation». In: (2018). DOI: https://arxiv.org/pdf/1810.04304.pdf (cit. on p. 11).

[5] Otkrist Gupta and Ramesh Raskar. «Distributed learning of deep neural network over multiple agents». In: *Journal of Network and Computer Applications* 116 (Aug. 2018), pp. 1–8. DOI: 10.1016/j.jnca.2018.05.003 (cit. on pp. 11, 18, 20, 21, 23, 26, 28, 36).

[6] *PySyft*. URL: https://github.com/OpenMined/PySyft (cit. on pp. 12, 29–31).

[7] Martın Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (cit. on p. 12).

[8] Jeon Joohyung and Joongheon Kim. «Privacy-Sensitive Parallel Split Learning». In: Jan. 2020, pp. 7–9. DOI: 10.1109/ICOIN48656.2020.9016486 (cit. on pp. 12, 21, 24, 25).

[9] Nicolas Papernot, Shuang Song, Ilya Mironov, Ananth Raghunathan, Kunal Talwar, and Úlfar Erlingsson. «Scalable Private Learning with PATE». In: *International Conference on Learning Representations (ICLR)*. 2018. URL: https://arxiv.org/abs/1802.08908 (cit. on p. 17).

[10] Vepakomma, Praneeth and Gupta, Otkrist and Swedish, Tristan and Raskar, Ramesh. «Split learning for health: Distributed deep learning without sharing raw patient data». In: (2018). URL: https://arxiv.org/abs/1812.00564 (cit. on pp. 18–20).

[11] Chandra Thapa, M.A.P. Chamikara, and Seyit Camtepe. «SplitFed: When Federated Learning Meets Split Learning». In: (Apr. 2020) (cit. on p. 21).

[12] Praneeth Vepakomma, Abhishek Singh, Otkrist Gupta, and Ramesh Raskar. «NoPeek: Information leakage reduction to share activations in distributed deep learning». In: *CoRR* abs/2008.09161 (2020). arXiv: 2008.09161. URL: https://arxiv.org/abs/2008.09161 (cit. on pp. 23, 43, 48).

[13] Yehuda Lindell. *Secure Multiparty Computation (MPC)*. Cryptology ePrint Archive, Report 2020/300. https://eprint.iacr.org/2020/300. 2020 (cit. on pp. 29, 44).

[14] Craig Gentry. «Fully homomorphic encryption using ideal lattices». In: *In Proc. STOC*. 2009, pp. 169–178 (cit. on pp. 29, 44).

[15] Cynthia Dwork and Aaron Roth. «The Algorithmic Foundations of Differential Privacy.» In: *Foundations and Trends in Theoretical Computer Science* 9.3-4 (2014), pp. 211–407. URL: http://dblp.uni-trier.de/db/journals/fttcs/fttcs9.html#DworkR14 (cit. on pp. 29, 45).

[16] Yann LeCun and Corinna Cortes. «MNIST handwritten digit database». In: (2010). URL: http://yann.lecun.com/exdb/mnist/ (cit. on pp. 31, 37).

[17] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE*. 1998, pp. 2278–2324 (cit. on p. 31).

[18] Sicong Liu, Anshumali Shrivastava, Junzhao Du, and Lin Zhong. *Better accuracy with quantified privacy: representations learned via reconstructive adversarial network*. Jan. 2019 (cit. on pp. 45, 47).

[19] Gábor J. Székely, Maria L. Rizzo, and Nail K. Bakirov. «Measuring and testing dependence by correlation of distances». In: *Ann. Statist.* 35.6 (Dec. 2007), pp. 2769–2794. DOI: 10.1214/009053607000000505. URL: https://doi.org/10.1214/009053607000000505 (cit. on pp. 48, 49).

[20] *NoPeekNN*. URL: https://github.com/TTitcombe/NoPeekNN (cit. on p. 50).

[21] Cohen G., Afshar S., Tapson J., and van Schaik A. «EMNIST: an extension of MNIST to handwritten letters.» In: (2017). URL: http://arxiv.org/abs/1702.05373 (cit. on p. 50).

[22] *COVID-19*. URL: https://github.com/muhammedtalo/COVID-19 (cit. on pp. 55, 59).

[23] Tulin Ozturk, Muhammed Talo, Azra Yildirim, Ulas Baloglu, Özal yıldırım, and U Rajendra Acharya. «Automated Detection of COVID-19 Cases Using Deep Neural Networks with X-ray Images». In: *Computers in Biology and Medicine* 121 (Apr. 2020). DOI: 10.1016/j.compbiomed.2020.103792 (cit. on p. 56).

[24] *Chest X-Ray Images (Pneumonia)*. URL: https://www.kaggle.com/paultimothymooney/chest-xray-pneumonia (cit. on p. 59).

[25] *Introduction to Federated Learning and Privacy Preservation using PySyft and PyTorch*. URL: https://blog.openmined.org/federated-learning-additive-secret-sharing-pysyft/ (cit. on p. 64).