Federated Learning Network: Training distributed Machine Learning models with the Federated Learning paradigm

Eduardo Yáñez Parareda
Máster en Ingeniería Informática

Félix Freitag

January 2021

To my wife Eva, and my daughter Daniela <3

Acknowledgments

Special mention to the project mentor, Félix Freitag, for helping during the whole project, testing the developments, contributing with a lot of ideas and make the whole project progress more easily.

FICHA DEL TRABAJO FINAL

Título del trabajo:	Federated Learning Network: Training distributed Machine Learning models with the Federated Learning paradigm
Nombre del autor:	Eduardo Yáñez Parareda
Nombre del consultor:	Félix Freitag
Fecha de entrega (mm/aaaa):	01/2021
Área del Trabajo Final:	Sistemas Distribuidos
Titulación:	Máster en Ingeniería Informática

Resumen del Trabajo:

Este proyecto tiene dos objetivos principales. Por un lado, el estudio del concepto de Federated Learning, el cuál fue acuñado hace 3 años por un equipo de ingenieros de Google. Por otro lado, se pretende desarrollar un software que sirva como herramienta para crear una red distribuida de dispositivos, que sea capaz de entrenar modelos de Machine Learning manteniendo la privacidad de los datos utilizados para esos entrenamientos.

El resultado final es un software funcional que cumple nuestros requisitos iniciales, y es capaz de aplicar Federated Learning en un entorno distribuido, lo que nos permite validar, de una forma práctica los conceptos iniciales del estudio. A lo largo de este proyecto, se presentan los conceptos más importantes de Federated Learning, así como algunos de los frameworks de software que están empezando a surgir a partir del mismo.

Abstract:

This project has two main goals. On one hand, the study of the concept of Federated Learning, which was coined 3 years ago by a team of engineers from Google. On the other hand, it is intended to develop a software which serves as a tool to create a distributed network of devices, capable of applying Federated Learning with different Machine Learning models.

The result is a functional software that meets our initial purpose and can apply Federated Learning in a distributed environment, allowing us to validate in a practical way, the initial concepts of study. Throughout this project, the most important concepts of Federated Learning are presented, as well as some of the software frameworks that are starting to emerge from it.

Keywords:

Federated Learning, Machine Learning, privacy, distributed computation.

Index

Index of figures

1. Introduction

1.1 Context and rationale

In 2017, several Google researchers published the article "Communication-Efficient Learning of Deep Networks from Decentralized Data" [1], which defined a new concept called Federated Learning.

This concept is based on the idea of increasing privacy when training Machine Learning models. It is normal when working on a new algorithm to collect a lot of data and train a model with that data until it is accurate enough. This may seem harmless, but if we look more closely at the process, we can see that most of the data used can affect the privacy of the people associated with it. This is where Federated Learning comes in, whose main objective is to avoid sharing data for the training of an artificial intelligence model.

Although there is a growing body of research works about Federated Learning, it seems that currently no standard components have been created to carry out a correct implementation. The final result of this project is to demonstrate that it is possible to implement a real solution, and show that the results obtained are acceptable for training Machine Learning models.

1.2 Goals

This project has several objectives:

1. Explore this new concept of Federated Learning and investigate as much as possible how effective the training of Machine Learning models is using this technique.
2. Use one of the existing frameworks to train a model.
3. Try to implement our own solution to see Federated Learning working in a real environment.
4. Draw conclusions about Federated Learning. That is, strengths, weaknesses or possible improvements.

1.3 Approach and methodology

At first there was nothing obvious about how to tackle this project. Since everything was new to me, the approach was to begin by researching as much as possible about Federated Learning, other existing projects, or implementations in the form of frameworks or libraries that would serve to develop a solution that would allow the proposed objectives to be met.

As the research progressed, some tools began to appear that seemed to serve our goal, but after some time working with them, it was seen that the only thing that could be achieved was to perform simulations of Federated Learning algorithms.

The simulations were not enough for what was initially intended, so it was decided to create a basic proof of concept that was adequate, as a starting point, to create a real solution.

When all the requirements are clear in a project, an obvious option may be to follow a *Waterfall* methodology, but in this case, we had to assume that everything could change from one day to the next, since we could find ourselves with some unsolvable problem or, on the contrary, we could discover something new that would help us to achieve our goal in a way that had not been foreseen until then.

Therefore, it was decided to follow a more flexible strategy that would allow to evolve, in an incremental way, from a proof of concept to a final product, by using an iterative process with which the scope of the project could be easily changed if necessary.

In conclusion, the steps that have been followed to accomplish the project have been:

1. Research of concepts and available resources.
2. Experimentation.
3. Creation of a proof of concept.
4. Implementation of a product from the proof of concept.
5. Validation of the obtained product.
6. Drawing up conclusions.

1.4 Project planning

The planning of this project has been defined considering the deliveries of the different PECs as project milestones. Therefore, there will be several intermediate deliveries on these dates:

PEC1: 22nd September 2020.
PEC2: 10th November 2020.
PEC3: 14th December 2020.
Final delivery TFM: January 4, 2021.

A Gantt chart of the resulting project planning is shown below:

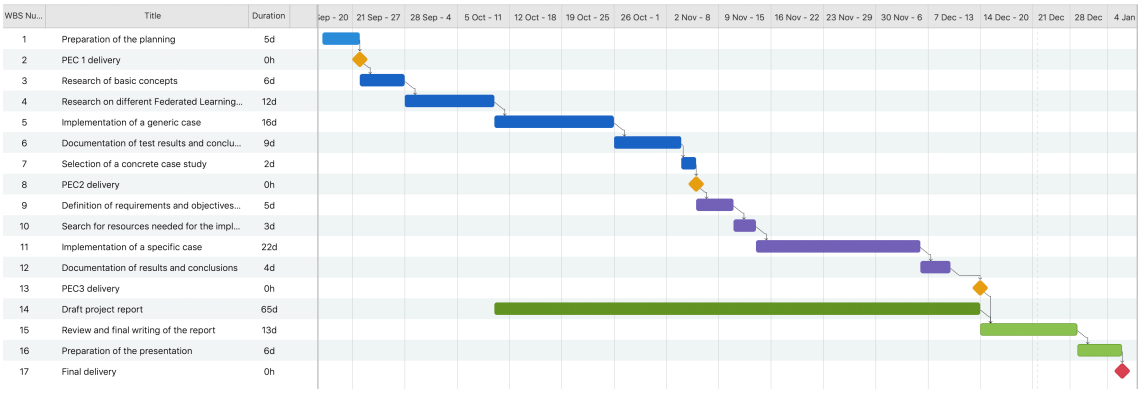


Figure 1: Gantt chart of the project planning

A more detailed view of the task within the chart can be viewed here:

WBS Number	Title	Duration	Start Date	End Date
1	Preparation of the planning	5d	17/ 9/2020, 14:30	21/ 9/2020, 17:45
2	PEC 1 delivery	0h	22/ 9/2020, 14:30	22/ 9/2020, 14:30
3	Research of basic concepts	6d	22/ 9/2020, 14:30	27/ 9/2020, 17:45
4	Research on different Federated Learning...	12d	28/ 9/2020, 14:30	9/10/2020, 17:45
5	Implementation of a generic case	16d	10/10/2020, 14:30	25/10/2020, 16:45
6	Documentation of test results and conclu...	9d	26/10/2020, 14:30	3/11/2020, 17:45
7	Selection of a concrete case study	2d	4/11/2020, 14:30	5/11/2020, 17:45
8	PEC2 delivery	0h	6/11/2020, 14:30	6/11/2020, 14:30
9	Definition of requirements and objectives...	5d	6/11/2020, 14:30	10/11/2020, 17:45
10	Search for resources needed for the impl...	3d	11/11/2020, 14:30	13/11/2020, 17:45
11	Implementation of a specific case	22d	14/11/2020, 14:30	5/12/2020, 17:45
12	Documentation of results and conclusions	4d	6/12/2020, 14:30	9/12/2020, 17:45
13	PEC3 delivery	0h	14/12/2020, 14:30	14/12/2020, 14:30
14	Draft project report	65d	10/10/2020, 14:30	13/12/2020, 17:45
15	Review and final writing of the report	13d	14/12/2020, 14:30	28/12/2020, 17:45
16	Preparation of the presentation	6d	29/12/2020, 14:30	5/ 1/ 2021, 17:45
17	Final delivery	0h	6/ 1/ 2021, 14:30	6/ 1/ 2021, 14:30

Figure 2: List of the high-level tasks planned for the project

The following lists and describes the different tasks with which this project is intended to be carried out. Each one is assigned an identifier, in order to refer to them in the Gantt:

1. **Preparation of the planning**: It includes the writing of this section itself, definition of the planning and definition of the objectives of the project in a broad way.
2. **PEC1 delivery**: Milestone corresponding to the official delivery of the PEC1.

3. **Research of basic concepts**: Search of information about the different concepts of Federated Learning, reading of articles, blogs, creation of a documentary base that serves as reference for the rest of the project.
4. **Research on different Federated Learning solutions**: Search for information in the reference documentation of the different frameworks with which to implement a generic solution, such as TensorFlow or Pytorch. The aim is to lay the foundation for implementing a neural network using this technique.
5. **Implementation of a generic case**: Implement a generic case in Python using one of the frameworks investigated in the previous section. The result will be a source code published in a public repository such as GitHub.
6. **Documentation of test results and conclusions**: The process followed to develop the implementation of the generic case, conclusions about the difficulties encountered, and results and response times will be written.
7. **Selection of a concrete case study**: Once we know how to implement a solution with Federated Learning, we will try to find a concrete case where to apply it or modify some of the parameters or the way to implement it.
8. **PEC2 delivery**: Milestone corresponding to the official delivery of PEC2, where the documentation generated during tasks 3 to 7 will be delivered.
9. **Definition of requirements and objectives of the specific case**: This task will consist of clearly defining the results and objectives of the specific case. What we expect to obtain or implement, what we want to prove, the restrictions we will impose, etc.
10. **Search for resources needed for the implementation**: Probably to carry out a different case you will need to install other software or use a different hardware or device. The objective of this task is to install and configure what is necessary to implement our specific solution.
11. **Implementation of a specific case**: Implement the specific case defined in the previous tasks. The result will be a source code published in the same repository as the generic implementation.
12. **Documentation of results and conclusions**: Task similar to number 6, in which the process followed to develop the selected concrete case, conclusions on the difficulties encountered, and results and response times will be written.
13. **PEC3 delivery**: Milestone corresponding to the official delivery of the PEC3, where the documentation generated during tasks 9 to 12 will be delivered.
14. **Draft project report**: This task will be started sometime during phase 2 (PEC2), and its result will be the draft report with all the documentation generated until that moment.
15. **Review and final writing of the report**: This task will finish reviewing, reordering, correcting, adding or deleting information to give as the result the final document of the report of the project.
16. **Preparation of the presentation**: As its name indicates, the result will be a document with the presentation of this project.
17. **Final delivery**: Final milestone corresponding to the official delivery of all the artifacts generated during the project.

1.5 Overview of products obtained

This project has allowed us to obtain several artifacts. Firstly, a documentation of the concepts and part of the existing software about Federated Learning (chaper 2).

We have also been able to create a basic version of a simulator that trains a Machine Learning model, for image prediction, based on the MNIST dataset. The simulator has been developed using TensorFlow Federated (chapter 3).

This was followed by our proof of concept, which allowed for training in the Federated Learning technique. Finally, we evolved the proof of concept towards a distributed system that can be managed from a web browser, capable of training different type of machine learning models using Federated Learning (chapter 4).

1.6 Chapters summary

1. **Introduction**: In this chapter we talk about the project itself, the reasons why it has been carried out, objectives, the methodology followed, high-level planning, and products achieved at the end.
2. **Federated Learning concepts**: Definition of the concepts that have been researched, and an overview of the main software libraries that can be used.
3. **TensorFlow Federated simulator**: The simulator implementation and its operation is described in this chapter.
4. **Design of a Federated Learning network**: This is the bulk of the report, which describes what was intended to be achieved with the proof of concept, how the final product was reached, the evaluation of the product itself, and possible improvements that can be made in the future.
5. **Conclusions**: It is a summary of lessons learned, and an overall assessment of the project.
6. **Glossary**: Brief definition of the most important concepts.
7. **Bibliography & citations**: Bibliography reference and citations.
8. **Annexes**: Installation & user's guide of the software produced.

2. Federated Learning concepts

At the beginning of the project, the first work was to gather information about Federated Learning concepts and try to understand the necessary details, strengths and weaknesses. To do this, a multitude of research articles were read, articles on specialized websites, and others on some less specialized websites, but all of them related to Federated Learning and the world of Machine Learning in general. The following chapters present the concepts that I believe are the most important to understand the context of the project.

2.1 What is Federated Learning?

Is a concept coined in the research article by McMahan et al. [1], which defines a technique or solution for training machine learning models in a distributed manner among several nodes, without sharing the original data used for training among those nodes. The original definition also specifies that there must be a central node that orchestrates this distributed training.

Data privacy, and regulations such as GDPR (General Data Protection Regulation), have been the main drivers for the birth of this technique. It is intended to prevent personal data from being sent over the network and from being stolen or manipulated by more or less malicious third parties.

To give an application example, Federated Learning can be used to improve the user experience (UX) in the use of certain mobile device, tablet or computer applications, such as text prediction or image categorization. With this technique, predictive models can be trained without sharing the data that users have on their devices (text messages, personal photos, etc.)

But Federated Learning is not only useful in this kind of scenarios. It can also be applied at the level of data centers in companies, hospitals, universities or different government institutions that cannot share their data because they are too sensitive.

The objective is to train models without sharing data, only the parameters calculated by the client nodes are shared.

2.2 The high-level algorithm

The broad outline of how Federated Learning works is as follows: The client nodes receive the current state of the model from a central node, then they train the model using their local dataset. The training of the client nodes is the same for all of them, in principle we use the SGD method (Stochastic Gradient Descent) [3] (or one of its derivatives such as mini-batch-SGD), as a function of cost of the neural network. The result will be new weights for the parameters of the model that finally will be send to the central node.

The central node receives the weights of the parameters calculated by each client node and carries out a mean (Federated Average) with which it updates the original model that it had previously sent to the clients. The use of Federated Average is what is proposed in the original article [1] that gave rise to the concept of Federated Learning.

In a cross-device scenario, as it is expected that the quantity of nodes client could be pretty big, not all the nodes will take part in the calculation at the same time, so from the total of the K nodes a quantity C is chosen randomly, which are those that in the following round they will do the calculations. In each round, other clients are chosen at random.

After all, the key is that the local data of the client nodes is never shared with any other node.

Algorithm steps:

1. The central node selects C random clients from the K nodes registered in the network.
2. The central node requests a training turn to the clients, sending the global model parameters (if any) to each client.
3. Each client performs a number of epochs of training predefined by the central node.
4. When a client finishes its training, it sends back the model parameters calculated locally to the central node.
5. When the central node has received all the model parameters from the client nodes, it calculates the average of the parameters and updates the model params.
6. Goes to step 1.

2.2.1 Hyperparameters

The *hyperparameters* of a Federated Learning training are the number of *epochs*, *learning rate* and *batch size* that will be used during the training.

If the nodes are able to perform different trainings, these parameters should be adjusted for each type of training. Also, if the system is advanced, it would be advisable to adjust these hyperparameters by each client, depending on the capacity of the client node, number of samples to train the model, etc.

2.3 Classification according to the type of client nodes

These are two terms used in Federated Learning to describe the type of nodes involved in the process (Feng et al. [2]).

2.3.1 Cross-device

We talk about cross-device when it comes to small devices such as cell phones, tablets, sensors, small computers (edge devices), etc. Figure 3 illustrates this case. In this case it

is expected that there may be thousands, hundreds of thousands or millions of devices participating in the training.
In this type of Federated Learning, it is also taken into account that the devices do not all have absolute availability, and that communications can be expensive, so it is tried that they use Wi-Fi connections, that they do the calculations while they are in standby mode, or when they are not being used by the user.

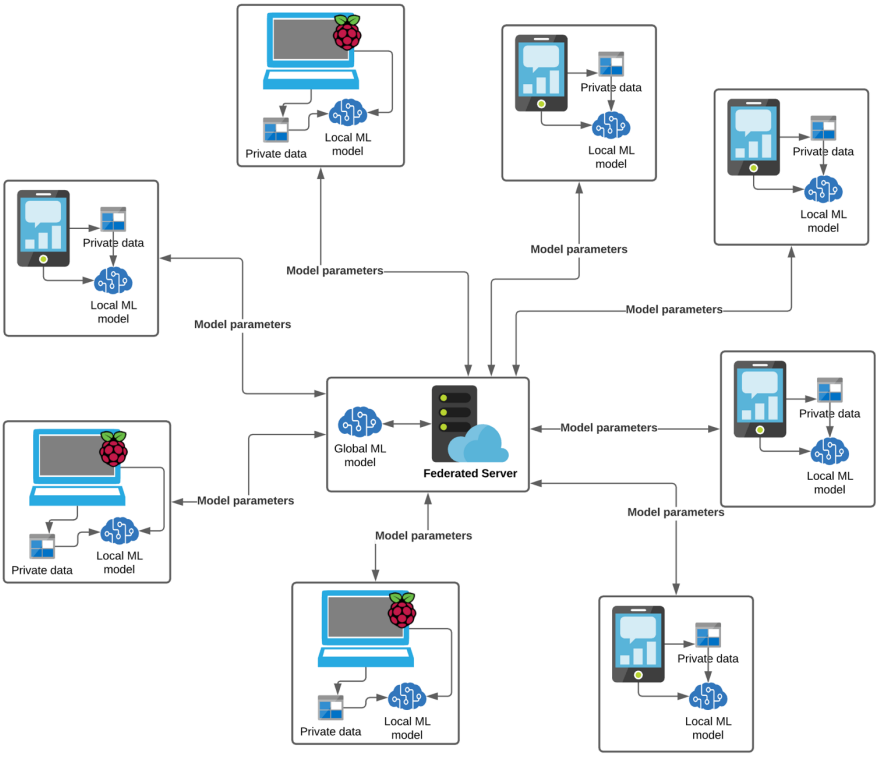


Figure 3: Cross-device scenario

Federated setting cross-device

It defines the scenario for applying federated learning when we are in cross-device.
The characteristics defined by McMahan et al [1] are:

- The data is non-IID. That is, the data used for training will not be independent of each other, and its distribution will not be the same either. For example, there will be times when the model training will be carried out by nodes in the same time zone or geographical area, or a client may have a large amount of data and the rest will not. All this will produce biases in the final model.
- Unbalanced data. Some devices will have a lot of data, and others will have very little.
- The data is highly distributed. The number of devices participating will be much higher than the average amount of data per node.
- Limited communications. Many devices may be off or have slow or expensive communications.

2.3.2 Cross-silo

The term cross-silo is used when instead of small devices, we talk about institutions or companies (Figure 4). Nodes are no longer small devices; they are possibly datacenters. In this case, it is expected that their number is not as large as in cross-device, that all nodes participate in the computation, and that the expected power of the nodes is much higher than in the previous case.

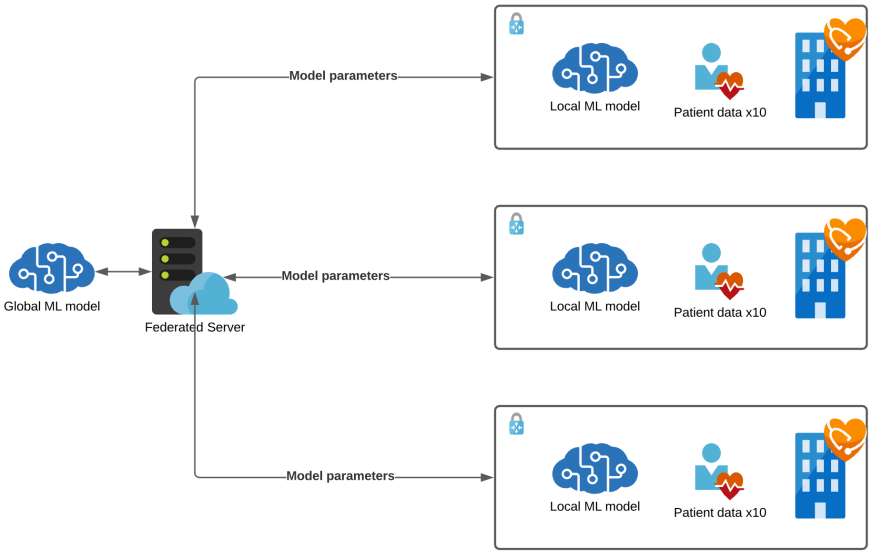


Figure 4: Cross-silo scenario

Federated setting cross-silo

The characteristics are:

- Customers are always available.
- Communications are not expensive, or customers can easily assume that cost.

2.4 Categorization according to data distribution

A dataset is made up of samples, each sample is made up of features or characteristics, and they can also be identified by customer. Depending on how the datasets of each client node are, and how they intersect between them, we can have three categories of Federated Learning [7].

2.4.1 Federated Learning horizontal

In this case the local datasets share the same type of features, but completely different user samples. For example, we may have two companies that do the same thing, collecting data from their customers. As they are the same type of business, they will share many of the features, but the intersection between their customers will be very low. The same thing happens in a cross-device scenario where a neural network is trained for text prediction. The users are all different, but the features of the data are common.

2.4.2 Federated Learning vertical

In this case the features are completely different, but the origins of the samples (clients) are more likely to match. For example, two entities from the same geographical area, one is dedicated to banking and the other to online sales. The objective could be to create a predictive model based on financial data and data on purchasing preferences, always preserving the identity of the customers, and without sharing the data between the two entities.

2.4.3 Federated Transfer Learning

It is the extreme case of vertical Federated Learning in which few features and few samples match. For example, a European bank and an Asian online sales company, the number of samples, due to the geographical distance, that coincide will be very low, and of course the features of their data will be very different. We try to train the model with the few samples and features that coincide, and then try to obtain predictions from the entities separately, with their own features.

2.5 Privacy and security

Although the main objective of Federated Learning is to keep the privacy of people, research such as that of Nasr et al. [4] and Zhu et al. [5] has shown that certain private data can be inferred from the models or calculated gradients. To avoid this, the study by Feng et al. [2] proposes a technique for the server to share an encrypted model, and for the client nodes to send their calculations also with noise. In this way, neither the clients can infer anything from the original model, nor the server from the clients' calculations.

2.6 Frameworks for Federated Learning

To date there are three main frameworks for conducting research with Federated Learning. Obviously, all three use Python as programming language, which is the de facto standard in machine learning:

- **TensorFlow Federated**: It is an opensource framework created by Google and based on TensorFlow. Mainly focused on the simulation of cross-device scenarios.
- **PySyft**: It is also an opensource library, oriented to security and privacy in Deep Learning. Initially based on PyTorch, although they plan to use it with TensorFlow. It has been created by the private company OpenMined.
- **IBM Federated Learning**: As its name suggests, is owned by IBM. It is not opensource but can be used for free. It provides tools to implement a real scenario.

2.7 Simulations

In a cross-device scenario we must expect the number of client nodes involved to be large (thousands or millions of them), so it is very difficult to implement a solution without

being able to simulate this type of scenario. How else are we going to do a test with 100,000 nodes, for example?

Therefore, what we do is simulate this type of scenario. A main process (the simulator) divides a special dataset (Federated dataset) in two, one that will be used for customer training, and another to evaluate the final model.

In addition, some functions are implemented that simulate the central server, and other functions that simulate the clients. Each one of the clients will use its own local dataset that is not shared by the rest, and once its calculations are made, it sends them, as if it were a real process, back to the entity that simulates the server. It aggregates the data received from the clients and repeats the process.

2.8 Federated datasets

It is an interesting concept (Caldas et al. [6]) from the point of view of researching possible implementations of a Federated Learning solution, mainly in a cross-device scenario. For machine learning there are different datasets to be able to perform training, they are datasets of images of people, animals, X-rays, texts, data provided by different sensors, etc. These datasets are very useful when trying different machine learning algorithms, and in particular deep learning to train neural networks.

In Federated Learning these traditional datasets do not serve as they are made. To be able to use them and to be able to simulate an implementation it is necessary to transform them previously because it is supposed that each client has its own data, so it is necessary that the transformation generates different subsets grouped by client identifier. Obviously in a real scenario there would not be any type of customer identifier, but in this case, it is necessary for the simulator process to be able to correctly distribute the dataset among the different processes that simulate the customers.

The LEAF project [6] provides several datasets already prepared to be used in different simulation processes:

- *Federated Extended MNIST (FEMNIST)*, based on the handwritten letters and numbers dataset. Used for the recognition of handwritten letters and numbers.
- *Shakespeare*, created from Shakespeare's plays, in which each customer is a character in a play, and the dataset is the texts recited by that character. Used for text generation.
- *Sentiment140*, used to evaluate the feelings of written messages, based on the emoticons written in Twitter tweets. Each Twitter account used is a different client.
- *CelebA*, based on CelebFaces Attributes, composed of images of famous people. Each person is a different user.
- *Reddit* contains comments and messages made by Reddit network users.

These are some of the statistics [6] of each of these Federated datasets, where we can see for example the number of nodes that we could use in a simulation (number of devices):

Name	Number of devices	Total of samples	Samples by device	
			Median	Average
FEMNIST	3.550	805.263	226,83	88,94
Sentiment140	660.120	1.600.498	2,42	4,71
Shakespeare	1.129	4.226.158	3.743,28	6.212,26
CelebA	9.343	200.288	21,44	7,63
Reddit	1.660.820	56.587.343	34,07	62,95

3. TensorFlow Federated simulator

The first weeks of the project, as mentioned in the previous chapter, were dedicated to understanding in a theoretical way how Federated Learning works. But one of the main goals was to put the theory into practice. Out of the frameworks named in section 2.6, I chose TensorFlow Federated because it seemed the most mature and had the most documentation around, either in the form of reference documentation [32], or through small workshops and videos [33] that taught how to work with it.

PySyft is also popular, but the documentation is still in a very primitive state. And IBM's proposal is very closed, so it is normal that it does not have many followers, although personally I think that the IBM solution is the closest one to the final product we have developed in this project.

3.1 TensorFlow Federated

Until the date of publication of this report, the only utility of this library is to make simulations of Federated Learning scenarios. This means that no real scenario is executed, but everything runs in the same machine, and in fact everything runs in the same process.

What TensorFlow federated provides is a way to simulate a network of nodes, where the central node requests the client nodes to do a training. The client nodes then perform that training, each with different data, return the calculated parameters of the Machine Learning model to the central node, and the central node averages those parameters, which are then used in subsequent training rounds. There is no communication between clients and the central node, in fact there are no such nodes, the central node is a function within the program, and the client nodes are calls to the same function within a for loop.

3.1.1 TensorFlow Federated annotations

Before starting to explain how the simulator works, it is necessary to explain what the TensorFlow Federated annotations are. In our simulator we use three types of annotations:

- **@tf.function**: This is not a Federated operation, it's just a Python decorator used by TensorFlow to indicate that the function will be used in a *tensor*. These functions will be used inside the Federated operations explained below.
- **@tff.federated_computation**: It is a Python decorator that makes the function mean that it is a Federated operation. This means that it may involve a network communication between two nodes in a network. It receives two parameters, first a function, and then a special type of data. This type of data must be previously defined with the class tff.FederatedType, which in turn receives two other parameters, a variable, which will be the data being transmitted, and a value tff.SERVER or tff.CLIENTS, which indicates where the transmitted data is used / stored. A function can be decorated with this annotation without using any

parameter. That means that the function is a Federated operation that doesn't implies communication.

- **@tff.tf_computation**: It's a decorator that tells TensorFlow Federated that the function will be called from a Federated operation (the ones annotated with tff.federated_computation).

(*) *tf* is the abbreviation for the Python package *tensorflow*. And *tff* is the abbreviation for the package *tensorflow_federated*.

3.2 The simulator

The goal of this simulator is to execute a Federated Learning training of a model that predicts if an image corresponds to a number from 0 to 9. It uses a special version of the MNIST dataset. It's special because it's divided by users. That means that the dataset is divided into as many subsets as users have participated in the contribution of images within the dataset. This is important for TensorFlow Federated to work properly, because it'll assign to each simulated client node a different subset depending on the user. This dataset is integrated into TensorFlow Federated, the framework itself downloads the dataset at the beginning of the process.

3.2.1 How it works

The guide to install and run the simulator can be found at Annex II, and the complete source code at Annex III.

Everything starts downloading the dataset:

```
8.  emnist_train, emnist_test = tff.simulation.datasets.emnist.load_data()
```

This line loads the training data and the test data. Then the simulation is managed by the *FederatedSimulator* class:

```
113.      # The Federated Lerning algorithm is an 'Iterative Process' which first initializes the server,
114.      # then run next_fn the number of rounds defined at the beginning of the simulation.
115.      federated_algorithm = tff.templates.IterativeProcess(
116.          initialize_fn=initialize_fn,
117.          next_fn=next_fn
118.      )
119.      federated_simulator.run_simulation(federated_algorithm)
```

First the *Federated IterativeProcess* is defined using an initialization function and the function that will be executed in each round of training. Both are Federated operations annotated with *@tff.federated_computation* annotation explained above. As explained in the Annex II, the parameters of the simulation (number of clients, batch size and number of rounds) can be changed in *simulation.py* file.

```
federated_simulator = FederatedSimulator(emnist_train, emnist_test, batch_format, create_keras_model,
                                         num_clients=10, batch_size=20, rounds=100)
```

Even if the dataset has more than 3000 different users, the simulation does not run over all of them, it selects randomly the number of clients set at *num_clients* variable. The function that performs this selection of data is in *FederatedSimulator.__build_federated_training_data*:

```
def __build_federated_training_data(self, training_data):
    client_ids = np.random.choice(training_data.client_ids, size=self.num_clients, replace=False)
    federated_training_data = [self.__preprocess(training_data.create_tf_dataset_for_client(x))
                               for x in client_ids
                               ]
    return federated_training_data
```

The initialization function *initialize_fn* initializes the server node with a Keras model, a loss function and some metrics, in this case a function to calculate the accuracy of the model.

```
def model_fn():
    """Creates the Keras model with a loss function, accuray as metric and the specification of the input data"""
    keras_model = create_keras_model()
    return tff.learning.from_keras_model(
        keras_model,
        input_spec=federated_training_data[0].element_spec,
        loss=tf.keras.losses.SparseCategoricalCrossentropy(),
        metrics=[tf.keras.metrics.SparseCategoricalAccuracy()]
    )


@tff.tf_computation
def server_init():
    """Initialization of the server model"""
    model = model_fn()
    return model.weights.trainable


dummy_model = model_fn()
```

Then, as explained for each round defined (100 by default) the *next_fn* function is called by the Federated process. The *next_fn* function is a high level Federated operation that is the core of the simulation:

```
@tff.federated_computation(federated_server_type, federated_dataset_type)
def next_fn(server_weights, federated_dataset):
    # Broadcast the server weights to the clients.
    server_weights_at_client = tff.federated_broadcast(server_weights)

        # Each client computes their updated weights.
        client_weights = tff.federated_map(
```

```
102.                client_update_fn, (federated_dataset, server_weights_at_client)

103.            )
104.
105.            # The server averages these updates.
106.            mean_client_weights = tff.federated_mean(client_weights)
107.
108.            # The server updates its model.
109.            server_weights = tff.federated_map(server_update_fn, mean_client_we
    ights)
110.
111.            return server_weights
```

First, it broadcasts the model parameters to the client nodes. Then computes the clients training and gathers the model parameters calculated by each of them. The next computation is to calculate the mean of those client parameters, and finally updated the server model weights, that will be broadcasted in the next round.

Depending on the number of clients and rounds it could take more or less time, also, if the computer doesn't have a good GPU, the simulation can be slow.

The last step of the simulation is to evaluate the accuracy of the model trained.

```
120.        federated_simulator.evaluate()
```

That calls FederatedSimulator.evaluate() function:

```
23. def evaluate(self):
24.     keras_model = self.__create_keras_model()
25.     keras_model.compile(
26.         loss=tf.keras.losses.SparseCategoricalCrossentropy(),
27.         metrics=[tf.keras.metrics.SparseCategoricalAccuracy()]
28.     )
29.     keras_model.set_weights(self.__server_state)
30.     keras_model.evaluate(self.__federated_test_data)
```

It evaluates the model with the last server weights calculated and uses the test dataset loaded at the beginning of the whole execution.

As result we can see something like this output:

```
2042/2042 [==============================] - 16s 8ms/step - loss:
1.5411 - sparse_categorical_accuracy: 0.6575

Process finished with exit code 0
```

With the final accuracy get by all the rounds of a simulation. In this case we got 65% of accuracy in the prediction.

3.3 Conclusions

It has been a bit disappointing that libraries like TenforFlow Federated or PySyft only serve to perform simulations, and do not provide anything to set up a real Federated Learning network. But on the other hand, they can be very useful if the goal is to test models with thousands of nodes, since it would be very complicated to mount a network of thousands of devices just to test a model of Machine Learning.

Even so, this type of simulation is still far from reality, since it does not take into account any of the problems intrinsic to a real distributed system, such as node failure, latency, communication failures, security, etc.

From the point of view of the development of this simulator, it has been pretty hard to modularize it properly because of the TensorFlow annotations needed to make it. These annotations cannot be put inside a class, because they need to be initialized with some external parameters, so in the end, they cannot be encapsulated. It seems that the goal of the programs made with TensorFlow Federated is to run them in *Jupyter notebooks*, or *Colab notebooks* (the Google's cloud version of *Jupyter* notebooks).

4. Design of a Federated Learning network

4.1 Goals

The main goal is to create an open-source project that provides a way to create networks for training Machine Learning models with Federated Learning. During this project, the code has been private, but it'll be released as opensource when this document is published.

The result will be a network of nodes that will train one or several models, but without sharing the data they have used for the training.

4.2 Architecture & design

Federated Learning limits the design of the network, as it specifies that there must be a central node that orchestrates the training of the model. That is, we will have a central node that will conduct the training, and that will receive the parameters of the trained local models, from the client nodes that have registered on the network.
Then, the network will have a typical *star* topology as we can see in Figure 5.

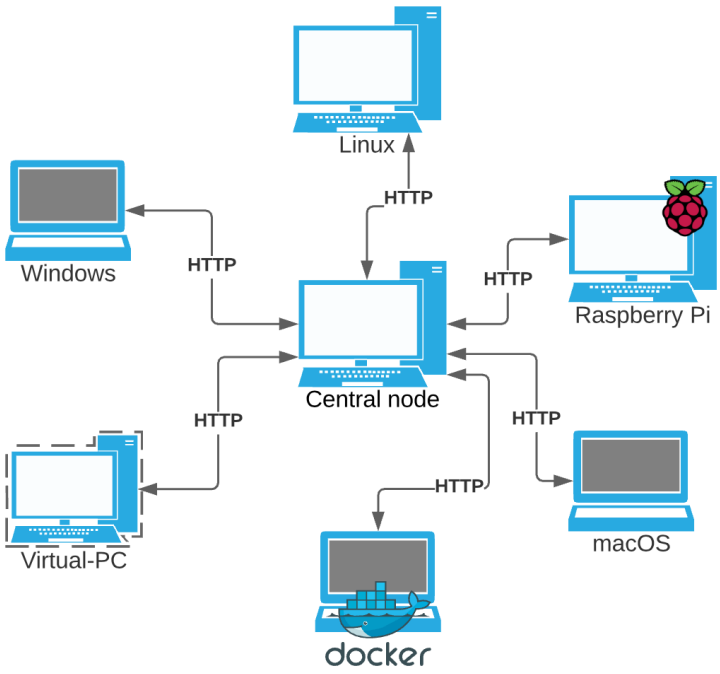


Figure 5: Basic network diagram

4.2.1 Nodes

The client nodes should be any device that can train a typical Machine Learning model, that is, where Python can be installed, and some of the libraries used for training the models, such as TensorFlow, Keras and PyTorch. We are talking about PCs with any

operating system, mobiles, tablets, virtual machines, Docker containers or edge computers like Raspberry Pi.

For this project we discarded the cell phones or tablets because it would not give time to develop a client for Android or iOS with the requirements we need.

As for the central node, it should not have any special requirements, as it will not perform any tasks that require much processing, although it should be able to manage many communications.

4.2.2 Code overview

There are two main modules implemented, on one hand, the *server* module, that represents the central node (see Figure 6).

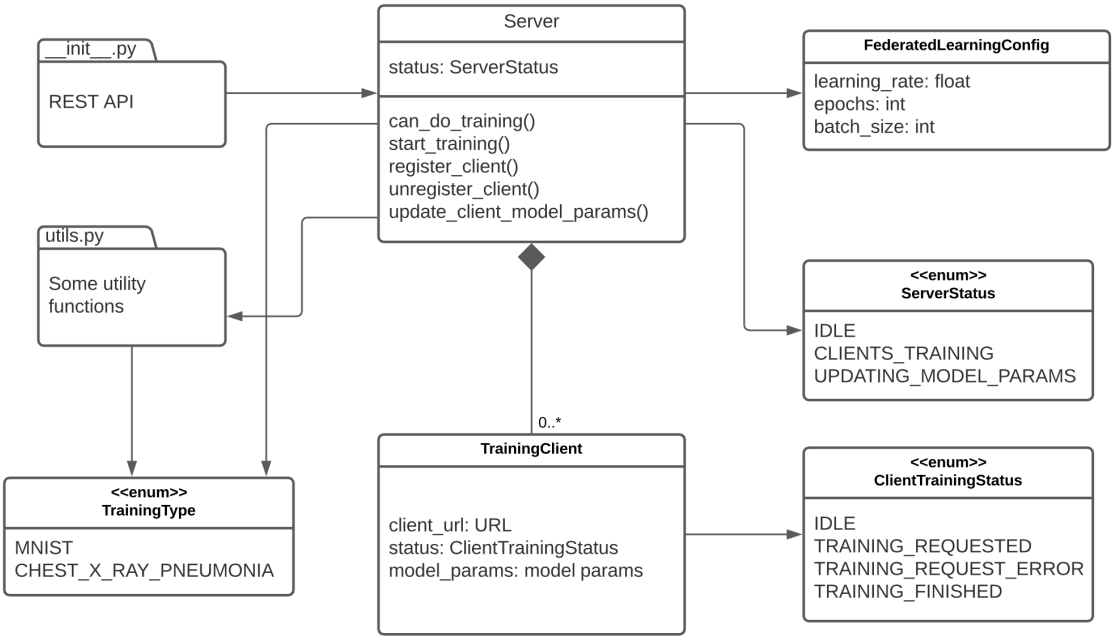


Figure 6: Class diagram of the server module

The main components are the *__init__.py* module that acts as controller and contains the REST API and the *Server* class that is responsible of all the logic and is always called from the REST API.

On the other hand, we have the *client* module, that is where the Machine Learning models are implemented (see Figure 7).

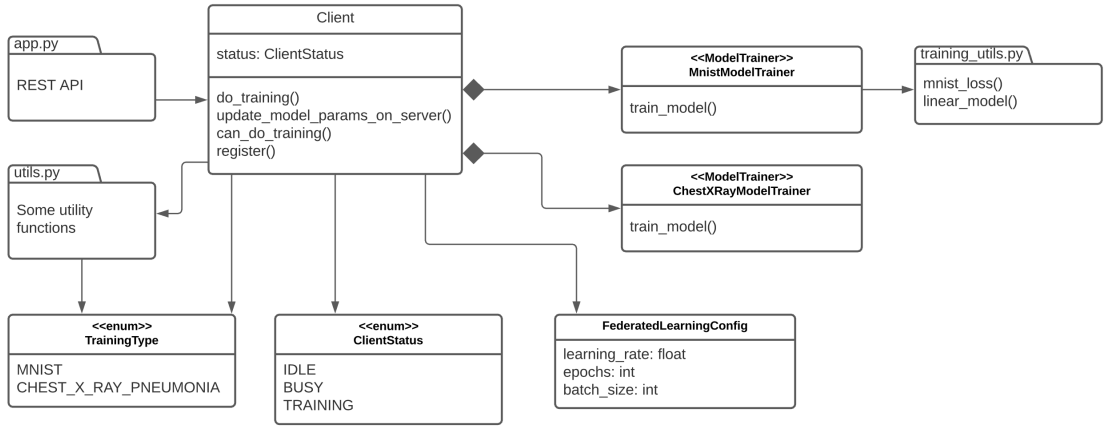


Figure 7: Class diagram of the client module

In the client module, the REST API calls the *Client* that depending on the type of training uses a different *ModelTrainer* implementation.

4.2.3 Communication protocol

Communications are performed using the HTTP protocol, through two different REST interfaces, one that will be implemented by the client nodes, and another that will be implemented by the central node. Ideally, HTTPS should be used, but this is left for future enhancement.

The central node has this set of basic REST operations:

- POST /client: Register a client in the network.
- DELETE /client: Delete client from the network.
- PUT /model_params: Update the calculated parameters after the training.

In the client nodes we have only one operation:

- POST /training: Train the model.

The choice of this protocol has been to facilitate the installation and configuration of the nodes, and to be able to access the central node from the Internet with any type of device.

4.2.4 How the whole system works

The most basic requirement for the system to work is that the central node is operational and at least one client node available for training. At the moment the client node starts, it tries to register in the network, and from that moment on it will be available for the central node to request trainings.

Having only one client node in this network does not make much sense. What is expected is that there will be several nodes registered to participate in the training of the model, because it is assumed that each client will not have much data to be able to perform efficient training.

Clients must know in advance what the IP address of the central node is, and at the time of the node's start-up, they will send a *POST /client* request (Figure 8), with their IP address so that the central node registers the client in its list of available clients for model training.

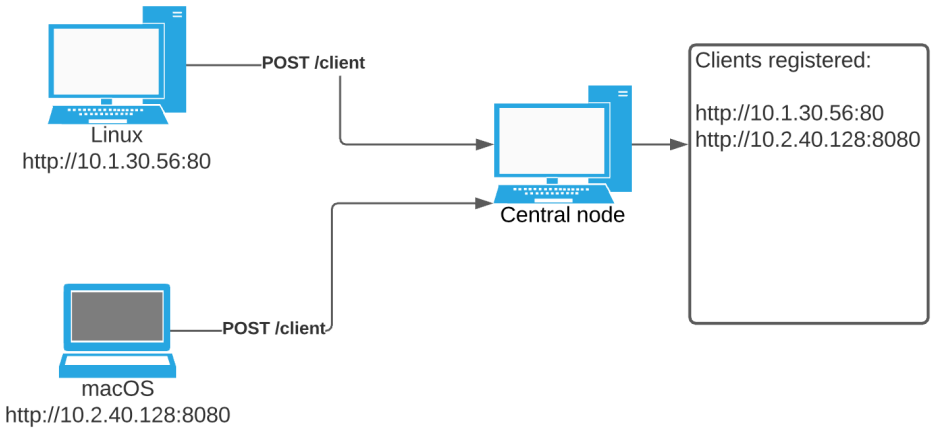


Figure 8: Node clients registering in the network

The central node will send a training request to the available, or not training client nodes. This request will be made asynchronously through the *POST /training* endpoint of the client nodes (Figure 9). It is at this point that the implementation of the corresponding Machine Learning algorithm starts. This request will send the averaged model parameters, if any, the type of training to perform (because a client node can be able to do several different trainings with different datasets), and the hyperparameters for the training (see section 2.2.1 Hyperparameters).

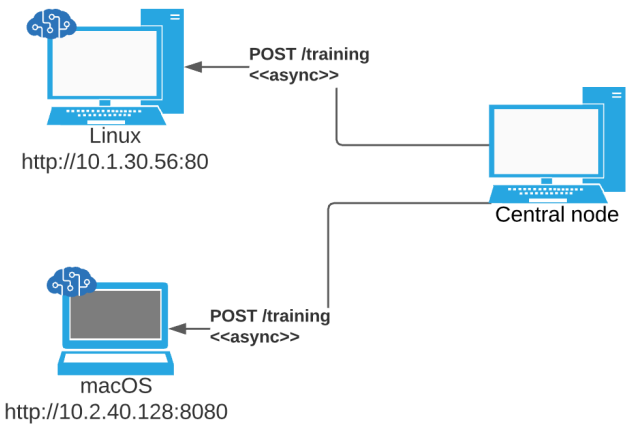


Figure 9: Central node requests training

When each client node finishes its training, it will send its response to the central node through the *PUT /model_params* endpoint (Figure 10).

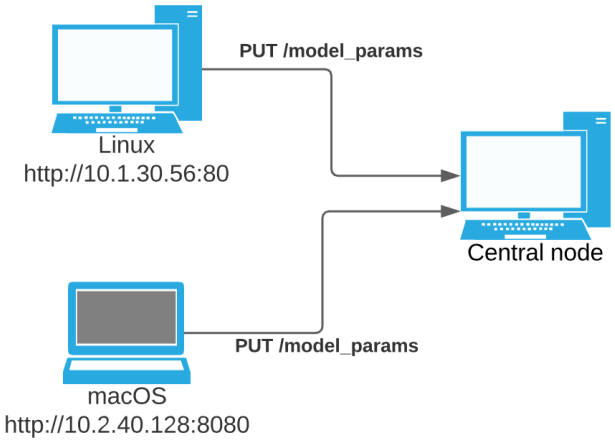


Figure 10: Clients send training results to central node

Once the central node has received all the parameters calculated by the client nodes, it will calculate the average of the parameter values, and store them for use in the next training round (Figure 11).

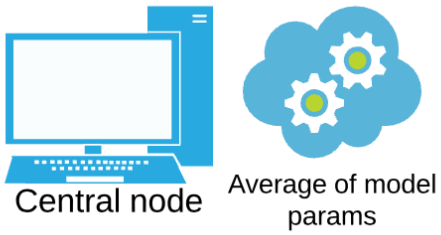


Figure 11: Central node computes average of the model parameters

4.2.5 Authentication

It is outside the scope of the project, and is left for future enhancements, but access to REST APIs, both client and central node, should be limited by some authentication process.

4.2.6 Data encryption

It is also outside the scope of the project. As already mentioned in point 2.5 Privacy and security, the parameters or features of the model should be encrypted before sending them to the central node, thus further protecting the privacy of the people related to the data.

4.3 Development environment

This Project has been developed on MacOS 10.15 and 11 versions. The language chosen has been Python version 3.8, that was the most stable version at the time.

To facilitate the development with Python, the first tool installed was *miniconda* (https://docs.conda.io/en/latest/miniconda.html). It's an environment and package management system that has been used to create virtual environments to avoid problems when using different versions of Python or the dependencies installed during the development.

There are several mainstream Machine Learning frameworks and libraries in Python, we have chosen two of the most known ones to prove that the final product can be used with any library. To implement the Machine Learning model used in this PoC we used PyTorch 1.7.0 and Fastai 2.1.5 libraries. On the other hand, during the second phase, the CNN model was implemented with TensorFlow/Keras.

Of course, the other main tool was Docker, used to create containers of the two kind of nodes.

Other general purpose software used has been IntelliJ as development.

4.3.1 Source code repository

The source code has been managed on GitHub in this repository:

- https://github.com/eyp/federated-learning-network

The source code of the PoC version can be found here:

- https://github.com/eyp/federated-learning-network/releases/tag/PoC

Whereas the code for the final version is at:

- https://github.com/eyp/federated-learning-network/releases/tag/v0.1.2

4.4 Proof of Concept implementation

The Proof of Concept (aka PoC) has been implemented in Python. The decision to choose this language has been made bearing in mind that Python is the most popular language for implementing Machine Learning algorithms.

For the development of the REST interface, Flask has been chosen as the development framework, because it is a very popular open-source framework with a large community behind it, as well as having good documentation.

The popularity of the language and the framework chosen has been a very important factor since there was no previous experience with any of them, so it was important that there was good documentation and support from different communities in order to solve possible problems during the development of the prototype.

4.4.1 Goals of the PoC

For this PoC the minimum requirements were:

- Have a central node that orchestrates the Federated Learning training.
- Be able to run several client nodes to train a Machine Learning model.
- Have one implementation of a Machine Learning algorithm.
- Implement the minimum REST interface to have everything working.

4.4.2 What was finally developed

When the deadline was reached all the requirements were properly implemented. Along the list of the initial requirements, the PoC added the possibility to be executed using Docker. Even if it was not in the main list, we thought that it would add lot of value and flexibility to the project, and would facilitate the testing on different machines, because with Docker we avoid the need to install Python and the dependencies of the application on the computers where the nodes are executed.

The developed prototype is able to run a central node and several clients on the same machine or on different ones, using Docker or just the command line. Although both the central node and the client nodes run web servers on different ports, no web pages have been added. It means that the status of the nodes and the training can only be seen through the command line console.

The nodes in the implemented network are able to train a very simple linear model that uses SGD (Stochastic Gradient Descent) as loss function, and a dataset provided by MNIST of samples of numbers 3 and 7 written by hand by lots of users.
The model tries to train a model to recognize if an image is a 3 or a 7.

Figure 12 shows some of the images provided by this dataset.

Figure 12: Images of a three and a seven character provided by MNIST dataset

4.4.3 PoC in action

When the central node starts running, we can see in a browser that it's running and its status (Figure 13):

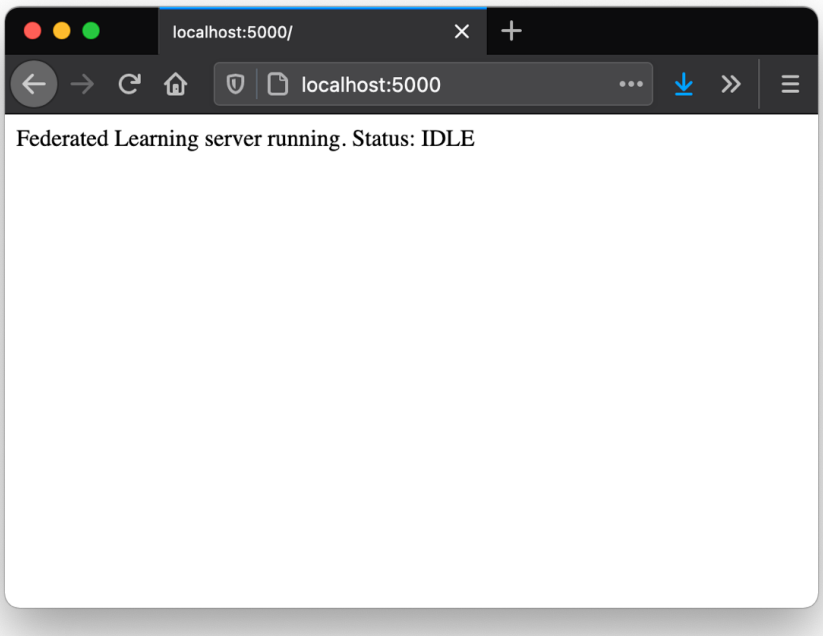


Figure 13: HTML page of the central node running on the Proof of Concept

Afterwards, if a client node is executed, we can see how it registers automatically on the central node:

```
Warning: SERVER_URL environment variable is not defined, using DEFAUL_SERVER_URL: http://127.0.0.1:5000
Registering in server: http://127.0.0.1:5000
Doing request http://127.0.0.1:5000/client
Response received from registration: <Response [201]>
Client registered successfully
 * Running on http://127.0.0.1:5003/ (Press CTRL+C to quit)
```

Figure 14: Console output of a client node registering on the Proof of Concept

The central node also shows that a new client has registered in the network:

```
Registering client: http://127.0.0.1:5003
Registering new training client [ http://127.0.0.1:5003 ]
Client [ http://127.0.0.1:5003 ] registered successfully
127.0.0.1 - - [22/Dec/2020 12:25:26] "POST /client HTTP/1.1" 201 -
```

Figure 15: Console output of the central node when a client registers in the network

A new training round can be launched using the browser at http://<central-node-IP-address>/training:

```
There are 1 clients registered
Requesting training to clients...
Requesting training to client http://127.0.0.1:5003/training
Updating model params from client: http://127.0.0.1:5003
New model params received from client http://127.0.0.1:5003
```

Figure 16: Console output of the central node requesting a training on the Proof of Concept

The client receives the hyperparameters of the training in the request (see next figure). It's also possible to see the evolution and the accuracy of the model after several training rounds:

```
Dataset ready to be used
Federated Learning config:
--Learning Rate: 1.0
--Epochs: 20
--Batch size: 256

Training started...
Accuracy of model trained at epoch 1 : 0.6667
Accuracy of model trained at epoch 2 : 0.6667
Accuracy of model trained at epoch 3 : 0.6667
Accuracy of model trained at epoch 4 : 0.6667
Accuracy of model trained at epoch 5 : 0.7222
Accuracy of model trained at epoch 6 : 0.7222
Accuracy of model trained at epoch 7 : 0.75
Accuracy of model trained at epoch 8 : 0.75
Accuracy of model trained at epoch 9 : 0.7778
Accuracy of model trained at epoch 10 : 0.7778
Accuracy of model trained at epoch 11 : 0.7778
Accuracy of model trained at epoch 12 : 0.7778
Accuracy of model trained at epoch 13 : 0.7778
Accuracy of model trained at epoch 14 : 0.7778
Accuracy of model trained at epoch 15 : 0.7778
Accuracy of model trained at epoch 16 : 0.7778
Accuracy of model trained at epoch 17 : 0.8333
Accuracy of model trained at epoch 18 : 0.8333
Accuracy of model trained at epoch 19 : 0.8611
Accuracy of model trained at epoch 20 : 0.8611
Training finished...
Response received from updating server model params: <Response [200]>
Model params updated on server successfully
```

Figure 17: Console output of the training result of a client for MNIST model

After a second round of training:

```
Dataset ready to be used
Federated Learning config:
--Learning Rate: 1.0
--Epochs: 20
--Batch size: 256

Training started...
Accuracy of model trained at epoch 1 : 0.8889
Accuracy of model trained at epoch 2 : 0.8889
Accuracy of model trained at epoch 3 : 0.8519
Accuracy of model trained at epoch 4 : 0.8519
Accuracy of model trained at epoch 5 : 0.8519
Accuracy of model trained at epoch 6 : 0.8519
Accuracy of model trained at epoch 7 : 0.8519
Accuracy of model trained at epoch 8 : 0.8519
Accuracy of model trained at epoch 9 : 0.8519
Accuracy of model trained at epoch 10 : 0.8519
Accuracy of model trained at epoch 11 : 0.8519
Accuracy of model trained at epoch 12 : 0.8519
Accuracy of model trained at epoch 13 : 0.8519
Accuracy of model trained at epoch 14 : 0.8519
Accuracy of model trained at epoch 15 : 0.8519
Accuracy of model trained at epoch 16 : 0.8519
Accuracy of model trained at epoch 17 : 0.8519
Accuracy of model trained at epoch 18 : 0.8519
Accuracy of model trained at epoch 19 : 0.8519
Accuracy of model trained at epoch 20 : 0.8519
Training finished...
Response received from updating server model params: <Response [200]>
Model params updated on server successfully
```

Figure 18: Console output of the result of a second round of training for MNIST model

And after a third round:

```
Dataset ready to be used
Federated Learning config:
--Learning Rate: 1.0
--Epochs: 20
--Batch size: 256

Training started...
Accuracy of model trained at epoch 1 : 0.8
Accuracy of model trained at epoch 2 : 0.8
Accuracy of model trained at epoch 3 : 0.8
Accuracy of model trained at epoch 4 : 0.8
Accuracy of model trained at epoch 5 : 0.8
Accuracy of model trained at epoch 6 : 0.8
Accuracy of model trained at epoch 7 : 0.8
Accuracy of model trained at epoch 8 : 0.8
Accuracy of model trained at epoch 9 : 0.8
Accuracy of model trained at epoch 10 : 0.8
Accuracy of model trained at epoch 11 : 0.8
Accuracy of model trained at epoch 12 : 0.8
Accuracy of model trained at epoch 13 : 0.8
Accuracy of model trained at epoch 14 : 0.8
Accuracy of model trained at epoch 15 : 0.8
Accuracy of model trained at epoch 16 : 0.8
Accuracy of model trained at epoch 17 : 0.8
Accuracy of model trained at epoch 18 : 0.8
Accuracy of model trained at epoch 19 : 0.8
Accuracy of model trained at epoch 20 : 0.8
Training finished...
Response received from updating server model params: <Response [200]>
Model params updated on server successfully
```

Figure 19: Console output of the result of a third round of training for MNIST model

All the clients train the model in parallel. It means that the central node requests the training to all of them, but it is not blocked, it can attend other requests. When all of the client nodes have sent their new calculated model parameters, the central node performs the average of the model parameters and stores it to be used in next training rounds.

4.5 First pre-release version

Once the Proof of Concept was finished, the purpose of the next planned phase was to refine and polish the product to get a more useful application.

To record a track of all the developments of this phase we used the *Issues* feature of GitHub (https://github.com/eyp/federated-learning-network), where all the development tasks have been registered, as it is shown in Figure 20:

Figure 20: List of issues created in GitHub's project

The red icon besides the title ( ) means the issue has been completed, and the green one ( ) means that is still a to-do task. So, in total 11 out if 23 issues were developed for this project, which led to version 0.1.2 presented in this document.

4.5.1 Features implemented

Among all the tasks there are some that stand out especially, and that are the ones that give a value to the product.

A dashboard

The Proof of Concept did not have any user interface to manage the network. Now a small dashboard has been implemented (Figure 21). Here we can see the status of the central node (called server in this version), and five client nodes registered in the network, waiting for training:

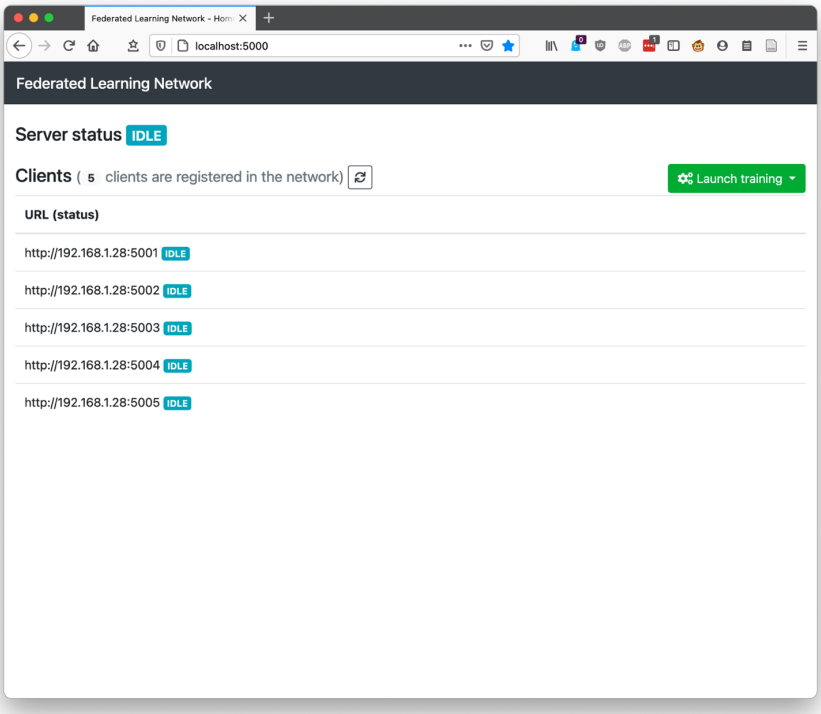


Figure 21: Network dashboard

It seems a simple feature but it is very useful because from there we can see the status of every node in the network and launch trainings. When the *Launch training* button is pressed, a dropdown with the available trainings is shown (Figure 22).

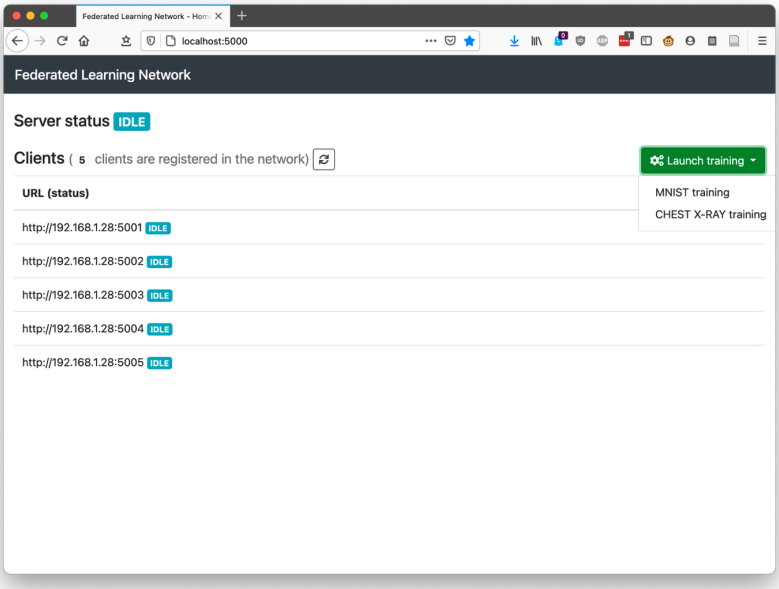


Figure 22: Dashboard showing the different type of trainings available

Figure 23 shows how the status changes when the training has been requested to the client nodes:

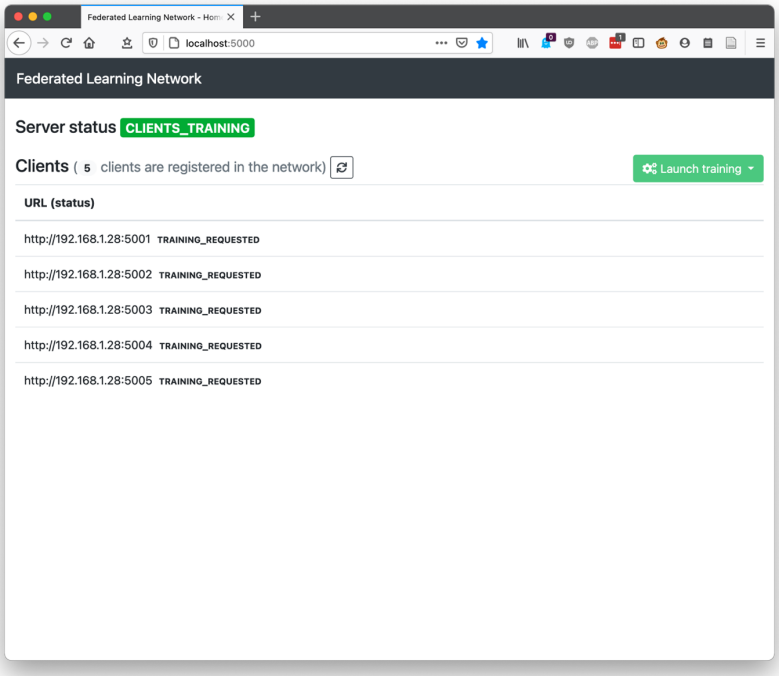


Figure 23: Status of the network when a training is requested

Or in the middle of a training, when some of the clients have already finished, but others are still working as it is shown in Figure 24.

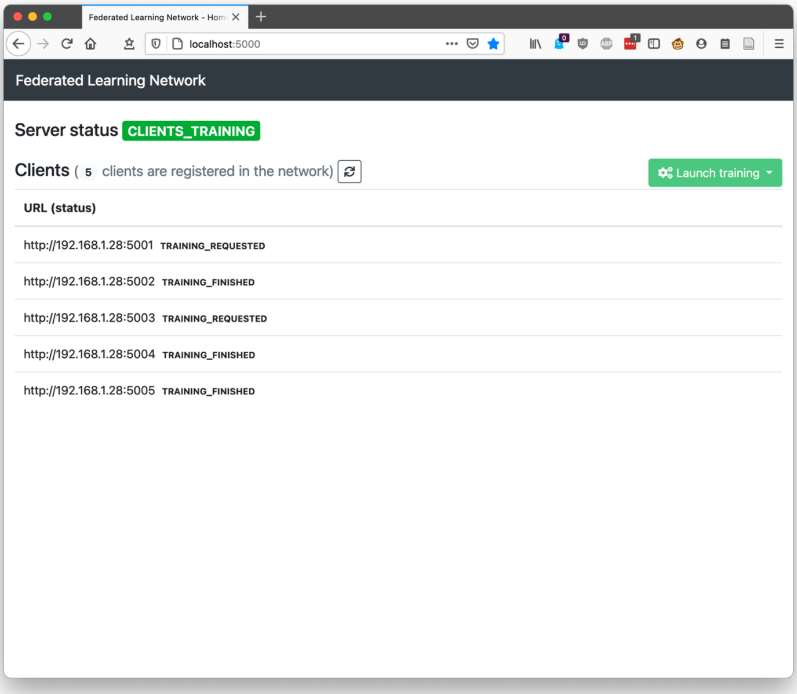


Figure 24: Dashboard when some clients have finished the training but others not

The Chest X-Ray Pneumonia algorithm

The PoC had only one algorithm implemented. A model which is able to predict if a number written by hand is a 3 or a 7. In the next version of the prototype we introduced a CNN (Convolutional Neural Network) algorithm capable of predicting, from lung radiographs (Figure 25), if a patient suffers from pneumonia.

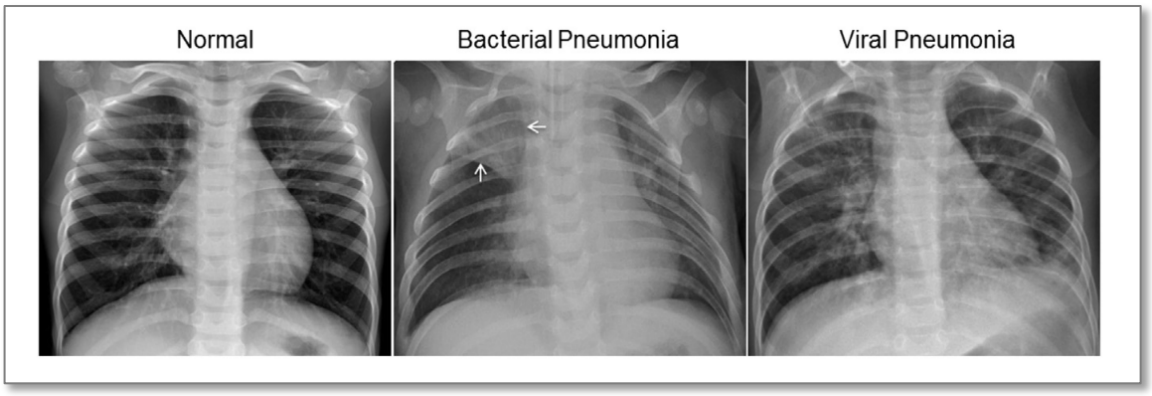


Figure 25: Examples of chest x-ray used in one of the models. Image extracted from [9]

For the implementation of this CNN model, we used TensorFlow/Keras library, and is based on an article of Varshita Ser [9]. It's a typical CNN made of five layers and a kernel of 3x3, that's is the most typical size used in this kind of algorithms.

The dataset ChestXRay2017 [10] is composed of these quantity of x-ray images:

	Normal	Pneumonia
Training (train folder)	1352	3887
Validation (test folder)	237	393

Our implementation uses the test folder for validation. Also, to simulate how would be a real client node we do not use all the images in the training inside a client node, instead, we choose 100 random images of each class (normal and pneumonia) from the training set, and 50 images of each class from the validation set. Each training round will select another random set of images, and of course, each client node selects its own random images. In a real scenario each client node should have its own dataset.

When a training is requested by the central node, we can see in the console output the results of the training (Figure 26). First, the client receives the request:

```
Request POST /training for training type: CHEST_X_RAY_PNEUMONIA
No weights found in the request
Federated Learning config:
--Learning Rate: 0.0001
--Epochs: 1
--Batch size: 2

Initializing ChestXRayModelTrainer...
Training started...
```

Figure 26: Console output of a training request for Chest X-Ray Pneumonia model

Then we can see that in the first training, the central node did not send any model params to the clients, and we can also see the validation accuracy achieved (50%) after the first round (Figure 27):

```
Using default model weights
Temporary dataset folder /federated-learning-network/node/tmp/chest_xray/ doesn't exist, creating it
Temporary folder for training: /federated-learning-network/node/tmp/chest_xray/tmptkk7jcli
Loading CHEST X-RAY IMAGES dataset...
Found 200 images belonging to 2 classes.
Found 100 images belonging to 2 classes.
10/10 - 2s - loss: 103.1614 - accuracy: 0.5000 - val_loss: 33.3936 - val_accuracy: 0.5000
Deleting content of temporary folder /federated-learning-network/node/tmp/chest_xray/tmptkk7jcli
Deleting temporary dataset folder /federated-learning-network/node/tmp/chest_xray/
```

Figure 27: Result of the execution of a first training round of Chest X-Ray Pneumonia model

In a second round of training, the client node receives initial parameters computed by the central node in the previous round. Now the validation accuracy increases up to 70% (Figure 28):

```
Using model weights from central node
Temporary dataset folder /federated-learning-network/node/tmp/chest_xray/ doesn't exist, creating it
Temporary folder for training: /federated-learning-network/node/tmp/chest_xray/tmp_hfjxg2c
Loading CHEST X-RAY IMAGES dataset...
Found 200 images belonging to 2 classes.
Found 100 images belonging to 2 classes.
10/10 - 2s - loss: 24.0590 - accuracy: 0.7000 - val_loss: 12.9599 - val_accuracy: 0.7000
Deleting content of temporary folder /federated-learning-network/node/tmp/chest_xray/tmp_hfjxg2c
Deleting temporary dataset folder /federated-learning-network/node/tmp/chest_xray/
```

Figure 28: Result of a second round of training of Chest X-Ray Pneumonia model

Possibility to use external datasets

The machine learning algorithm implemented in the PoC version downloads its own dataset using *Fastai* library. Now, for the new Chest X-Ray Pneumonia algorithm it's possible to use an external dataset that can be configured with the variable GLOBAL_DATASETS defined in client/config.py file. Furthermore, if the client node is executed as a Docker container, this feature can be used passing an external volume to the container. Everything is explained in the user's guide (see Annex I).

4.6 Possible use cases

What's the meaning of working on a software if nobody can use it? That's the question I would like to answer in this section for this project in particular.

I think Federated Learning can be useful in different scenarios, for example:

- Machine Learning models applied for *rare diseases*. When we think or read something about AI applied to medicine, we always have in mind popular diseases. Of course, that's very useful, but probably Hospitals, Medical and Research centers have lot of samples to train those models. But what happens when that's not the case? For rare diseases, these institutions probably don't have enough data, and they cannot (or at least the shouldn't) share that information with other organizations. Each organization could use its own dataset to train a global model.
- Models applied on internet navigation information or emails, for example to create spam filters, anti-phishing models, etc. Nobody likes to share that information with external companies, we all know what happens with that data. Training these kinds of models locally, without sharing the data will help to keep the privacy of people.

- In general, any kind of collaboration between different organizations to produce prediction models. They could be universities, schools, financial institutions or banks, small retail business, governments, etc.

4.7 Assessment

Once the PoC of the software finished it is time to evaluate it. In order to do it, we have proposed several metrics that we can give another perspective about the product.

The tests to evaluate the system have been made on several environment and operating systems: MacOS 10.15, MacOS 11, Ubuntu 18.04.5 LTS, Debian 10 and Raspbian (Debian 10 based version) for the Raspberry Pi devices.

Ease of use

On one hand, since the target users of Federated Learning Network are specialized people, we think we have provided some mechanisms that make it easy to use it. A very simple user interface to manage the trainings, and Docker containers to avoid installation issues. Of course, instead of using Docker, everything can be installed manually and run it through the command line.

On the other hand, there are a lot of improvements that need to be made in this aspect. For example, the accuracy get after a client node trains its local model is not send to the central node, so it is not shown on the dashboard.

One of the efforts made after the PoC version was to simplify the way to run the client nodes. There were lot of things in that version that added complexity to use run it, but it has been simplified a lot, and now only a couple of environment variables are needed to make it work.

Flexibility

This was one of the goals while developing the system. At the beginning, for the Proof of Concept, the first model provided was developed using PyTorch because I knew it better than TensorFlow, but with the second version, when a model based on TensorFlow/Keras was introduced, it has been proven that the system is framework agnostic.

In fact, the way it's been developed gives the possibility to add new Machine Learning algorithms using any library. The main class of the client node *Client* at *client.py* file doesn't depend on any Machine Learning library, it uses the classes that implement the algorithms, in this case *MnistModelTrainer* and *ChestXRayModelTrainer*.

Of course, it is not so simple to add a new model, besides adding a new class for the new model, some methods in the system must be adapted, but they are easy to localize and change.

Security

This is the weakest point of the system, there is not security at all implemented. Now, a node that knows the central node IP address can connect to the network without any issues, and that could be a problem because an evil computation could manipulate the weights of the global weights used for the trainings.

Also, probably the parameters of the model should be ciphered when transmitted to the central node to provide a complete privacy on the local dataset used in client nodes.

Adding security is proposed as a future work in section 4.8.

Portability

The use of Python as a programming language, and the ability to deploy the nodes using Docker makes the system very portable. We have proven it running on nodes with different operating systems as we have already mentioned at the beginning of this chapter.

However, we have experienced several problems on some environments. For example, PyTorch has proven to be a problem for running nodes on Raspberry Pi. It is an effort beyond the scope of this project to install PyTorch on these devices because they do not have binaries available for it. We have only found references from people who have been able to install older versions of PyTorch, but not the versions we needed. So finally, to prove that it works in these *edge* computers what has been done is to remove the PyTorch based model and leave only the one based on TensorFlow/Keras.

Testability

It seems that it is easy to test the system, but only one side of it. Connection to the network, adding nodes dynamically, requesting trainings to client nodes, using different type of devices or operating systems, is relatively easy thanks to the use of Docker containers.

The bad side of the testing is about adjusting the implemented Machine Learning algorithms. Since the parameters of the models cannot be changed through any user interface, the only way to adjust them is changing the source code directly. The problem doing this, is that the Docker images used to run the containers must be re-created, and that's a slow process.

If the client nodes are run on a local installation, it's faster to change the code and run them again, but to make these tests faster it's recommended to run several clients on the same machine.

A very useful improvement could be adding the possibility to change some model training parameters from the dashboard.

Accuracy of the models

This is a point that depends completely on the models implemented, and the data used for training the models. For the MNIST model, even if it is a very simple linear implementation, not based on CNN, we have got accuracies up to 96% or 98% in several tests, but usually they are around 93%-95%.

Unfortunately, for the model that predicts if a patient has pneumonia (*Chest X-Ray Pneumonia*) the accuracies obtained has been very low, and very irregulars. This is due to the fact that this kind of models need more *epochs* and therefore they also need more data to train and test.

4.8 Future work

There is a lot of room for improvements or new features in the Federated Learning Network project. Some of them can be found in the own Issues list in the GitHub project. Since the project is open to everyone, I believe that it will invite other people to collaborate and contribute with new ideas or improvements in the future.

New features that could be added are:

- Improving the functionalities of the dashboard introducing a more advanced frontend framework like React or Vue.
- Training random clients in each round.
- Adding security features like HTTPS, transmission of encrypted model weights so the central node knows nothing about the original client calculated weights.
- Add a database to the nodes to keep the information about the nodes registered in the network and the models trained.
- Merge the nodes (central node and client) into one, so all the nodes can act as central node or client node.
- Add the possibility to load a picture or sample through the dashboard to test a model.
- Add more Machine Learning algorithms.
- Add authentication by provided token + IP address, for example.
- Allow to change model training parameters through the dashboard.

Some projects that could be done from this could be:

- Develop an Android client node.
- Develop an iOS client node.
- Develop client nodes for browsers using frameworks like *TensorFlow.js*.
- Create a real network of edge-devices or mobile phones to perform real trainings.

5. Conclusions

5.1 About Federated Learning

In the last 10 years the Machine Learning algorithms have reached an incredible popularity, and this has been due to the facility that has been in this time to create datasets that facilitate the training of these artificial intelligence algorithms. This can be seen as something positive, but a very negative side is the use and distribution of people's private data. It is in this point where I believe that Federated Learning has a lot to contribute.

Some other conclusions about Federated Learning that can be answered now, after being worked in this project are:

- It will (or at least it should) evolve to a more distributed model, instead of relying on a central node that controls the whole network.
- Since it doesn't need a cloud infrastructure, it will take some of the power or importance away from big internet corporations like Google, Amazon or Facebook, since other less powerful companies will be capable to create models without having access to as much data as these giants have.
- Given the increasing importance of people's privacy, this paradigm has no choice but to start being adopted by more and more companies, so I believe that in the next few years more and more effective solutions will start to appear.
- On the negative side, I believe that by not being able to train models with a lot of data, it will be more difficult to reach the precision that is achieved with traditional training, but I am sure that this will change in the near future.

5.2 Personal conclusions

Having worked in this project has given me the possibility of learning many new concepts which I don't usually work with. It is an area within AI that is expected to be very important in the very near future and being part of it means a lot to me on a personal level.

From the technical point of view, I have been able to develop a project using languages and tools practically unknown to me as Python, or libraries related to Machine Learning as PyTorch or TensorFlow. In addition, I have had the opportunity to use quite advanced concepts of Python to be able to implement the final solution It has not been an easy journey, but on the other hand it has been very interesting and has given me the possibility of contributing my work to the opensource world (*).

I believe the objectives of the project have been more than fulfilled. As it was a research project of new concepts, the mentor of the project and I were not clear how far we could go, but as progress has been made, I think we can be happy with the results. The initial goals were:

1. Investigate about Federated Learning: Done.
2. Use one of the existing frameworks to train a model: Done, a small simulator has been developed with TensorFlow Federated.
3. Try to implement our own solution: Done, Federated Learning Network is a functional system, although it has room for adding lot of improvements.
4. Draw conclusions about Federated Learning: Done (section 5.1).

As for the project planning, everything went quite well. The truth is that the second phase was very open as to what it was intended to do, but it has been possible to meet the planned dates. There has been no need to modify the planning at any time

Obviously, I would have liked to develop all the ideas I had in mind when starting the second phase of the project, and to result in a much more complete system with more features, but the time was not enough for everything. On the other hand, I had in mind to work on something that would be interesting and motivating from a personal point of view, and that would end up with something that you could continue with once the project was finished. This, I think it's something that I've also achieved. The project is still alive in a public platform like GitHub, and I hope to keep working on it as one of my side projects.

(*) https://github.com/eyp/federated-learning-network

6. Bibliography & citations

[1] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, y B. A. y Arcas, «Communication-Efficient Learning of Deep Networks from Decentralized Data», *arXiv:1602.05629 [cs]*, feb. 2017, Accessed: sep. 17, 2020. [Online]. Available at: http://arxiv.org/abs/1602.05629.

[2] Y. Feng *et al.*, «A Practical Privacy-preserving Method in Federated Deep Learning», *arXiv:2002.09843 [cs, stat]*, ago. 2020, Accessed: oct. 13, 2020. [Online]. Available at: http://arxiv.org/abs/2002.09843.

[3] «Stochastic gradient descent», *Wikipedia*. oct. 12, 2020, Accessed: oct. 13, 2020. [Online]. Available at: https://en.wikipedia.org/w/index.php?title=Stochastic_gradient_descent&oldid=983180780.

[4] M. Nasr, R. Shokri, y A. Houmansadr, «Comprehensive Privacy Analysis of Deep Learning: Passive and Active White-box Inference Attacks against Centralized and Federated Learning», *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 739-753, may 2019. [Online] Available at: 10.1109/SP.2019.00065.

[5] L. Zhu, Z. Liu, y S. Han, «Deep Leakage from Gradients», *arXiv:1906.08935 [cs, stat]*, dec. 2019, Accessed: oct. 13, 2020. [Online]. Available at: http://arxiv.org/abs/1906.08935.

[6] S. Caldas *et al.*, «LEAF: A Benchmark for Federated Settings», a*rXiv:1812.01097 [cs, stat]*, dec. 2019, Accessed: sep. 27, 2020. [Online]. Available at: http://arxiv.org/abs/1812.01097.

[7] Q. Yang, Y. Liu, T. Chen, y Y. Tong, «Federated Machine Learning: Concept and Applications», *arXiv:1902.04885 [cs]*, feb. 2019, Accessed: oct. 13, 2020. [Online]. Available at: http://arxiv.org/abs/1902.04885.

[8] Varshita Ser. Towards Data Science. «Beginner's guide to building Convolutional Neural Networks using TensorFlow's Keras API in Python». 4th Sep 2020. Accessed 29th November 2020. [Online]. Available at: https://towardsdatascience.com/beginners-guide-to-building-convolutional-neural-networks-using-tensorflow-s-keras-api-in-python-6e8035e28238

[9] Daniel S. Kermany, Michael Goldbaum,Wenjia Cai, ..., M. Anthony Lewis,Huimin Xia, Kang Zhang, «Identifying Medical Diagnoses and TreatableDiseases by Image-Based Deep Learning». 22nd Feb 2018. Accessed 29th November 2020. [Online]. Available at: https://www.cell.com/action/showPdf?pii=S0092-8674%2818%2930154-5

[10] Kermany, Daniel; Zhang, Kang; Goldbaum, Michael (2018), «Labeled Optical Coherence Tomography (OCT) and Chest X-Ray Images for Classification», Mendeley Data, V2, doi: 10.17632/rscbjbr9sj.2

[11] Jeremy Howard & Sylvain Gugger. «Deep Learning for Coders with fastai & PyTorch», O'Reilly.

[12] Aurélien Géron. «Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow, 2nd Edition», O'Reilly.

[13] Python documentation. [Online]. Available at: https://www.python.org/doc

[14] Kaggel. Chest X-Ray images (Pneumonia). Accessed: dec, 3, 2020. [Online]. Available at: https://www.kaggle.com/paultimothymooney/chest-xray-pneumonia/
[15] Flask documentation. Pallets Software. [Online]. Available at: https://flask.palletsprojects.com
[16] Théo Ryffel, «Federated Learning in 10 lines». *OpenMined Blog*, Mar. 01, 2019. Accessed Sep. 21, 2020 [Online]. Available at: https://blog.openmined.org/upgrade-to-federated-learning-in-10-lines/.
[17] S. Tijani, «Federated Learning: A Step by Step Implementation in TensorFlow» *Medium*, Sep. 09, 2020. Accessed Sep. 19, 2020. [Online]. Available at: https://towardsdatascience.com/federated-learning-a-step-by-step-implementation-in-tensorflow-aac568283399 ().
[18] Synced, «Federated Learning: The Future of Distributed Machine Learning» *Medium*, Feb. 14, 2019. Accessed Sep. 21, 2020. [Online] Available at: https://medium.com/syncedreview/federated-learning-the-future-of-distributed-machine-learning-eec95242d897.
[19] Saheed Tijani, «Federated Learning: Why, What and How?» *LinkedIn*, Apr. 14, 2020. [Online]. Accessed Sep. 21, 2020. Available at: https://www.linkedin.com/pulse/federated-learning-why-what-how-saheed-tijani/.
[20] David Gilmore, Philip Stubbings, Davis Wilkinson, Eitan Rovero-Shein. «Federated Learning for Credit Scoring» *OpenMined Blog*, May 26, 2020. [Online]. Accessed Oct. 01, 2020. Available at: https://blog.openmined.org/federated-credit-scoring/.
[21] Yang Qiang, «Federated AI, as explained by WeBank's Yang Qiang» *DigFin*, Aug. 31, 2019. [Online]. Accessed Sep. 21, 2020. Available at: https://www.digfingroup.com/federated-ai/.
[22] Daniel Ramage, Stefano Mazzocchi. «Federated Analytics: Collaborative Data Science without Data Collection», *Google AI Blog*. [Online]. Accessed Sep. 19, 2020. Available at: http://ai.googleblog.com/2020/05/federated-analytics-collaborative-data.html.
[23] Theo Ryffel, Andrew Trask, Morten Dahl, Bobby Wagner, Jason Mancuso, Daniel Rueckert, Jonathan Passerat-Palmbach, «A generic framework for privacy preserving deep learning» *arXiv:1811.04017 [cs, stat]*, Nov. 2018, Accessed: Oct. 10, 2020. [Online]. Available: http://arxiv.org/abs/1811.04017.
[24] Shiqiang Wang, Tiffany Tuor, Theodoros Salonidis, Kin K. Leung, Christian Makaya, Ting He, Kevin Chan. «Adaptive Federated Learning in Resource Constrained Edge Computing Systems», *arXiv:1804.05271 [cs, math, stat]*, Feb. 2019, Accessed: Oct. 10, 2020. [Online]. Available: http://arxiv.org/abs/1804.05271.
[25] P. Kairouz *et al.*, «Advances and Open Problems in Federated Learning», *arXiv:1912.04977 [cs, stat]*, Dec. 2019, Accessed: Sep. 29, 2020. [Online]. Available: http://arxiv.org/abs/1912.04977.
[26] V. Smith, C.-K. Chiang, M. Sanjabi, and A. Talwalkar, «Federated Multi-Task Learning», *arXiv:1705.10467 [cs, stat]*, Feb. 2018, Accessed: Sep. 27, 2020. [Online]. Available: http://arxiv.org/abs/1705.10467.

[27] Chris J. Preimesberger. «Six Data Points You Should Know about Federated Machine Learning», *eWEEK*. 7th Feb 2020. [Online] Accessed Sep. 21, 2020. Available at: https://www.eweek.com/big-data-and-analytics/six-data-points-you-should-know-about-federated-machine-learning.

[28] Zheng Chai, Ahsan Ali, Syed Zawad, Stacey Truex, Ali Anwar, Nathalie Baracaldo, Yi Zhou, Heiko Ludwig, Feng Yan, Yue Cheng, «TiFL: A Tier-based Federated Learning System», *arXiv:2001.09249 [cs, stat]*, Jan. 2020, Accessed: Oct. 13, 2020. [Online]. Available: http://arxiv.org/abs/2001.09249.

[29] G. F. Volpi, «Should I inlcude the intercept in my machine learning model?», *Towards Data Science*, Feb. 11, 2020. [Online]. Accessed Oct. 01, 2020. Available at: https://towardsdatascience.com/should-i-inlcude-the-intercept-in-my-machine-learning-model-2efa2a3eb58b.

[30] «IBM/federated-learning-lib: A library for federated learning (a distributed machine learning process) in an enterprise environment». [Online]. Accessed Sep, 21, 2020. Available at: https://github.com/IBM/federated-learning-lib.

[31] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, Karn Seth, «Practical Secure Aggregation for Privacy Preserving Machine Learning», 281, 2017. Accessed: Sep. 27, 2020. [Online]. Available: https://eprint.iacr.org/2017/281.

[32] TensorFlow Federated. Google. https://www.tensorflow.org/federated

[33] Adria Gascon, Marco Gruteser, Peter Kairouz, Zheng Xu, Rainier Aliment, Jennifer Ye. «Federated Learning Workshop using TensorFlow Federate». Jul. 31st, 2020. Google. [Online]. Accessed: Oct. 2020. Available at: https://events.withgoogle.com/demostutorials-workshop-on-federated-learning-and-analytics-2020/

Annex I: Federated Learning Network installation & user's guide

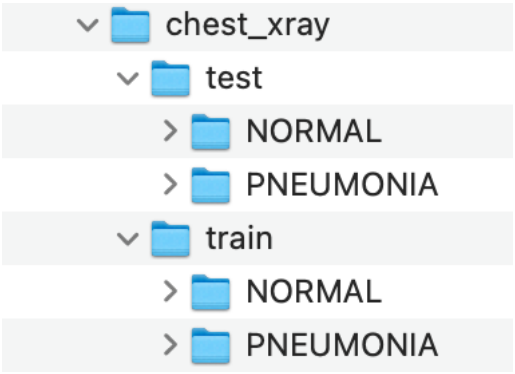
Introduction

There are two options for running the nodes in the network, using Docker to create containers for each kind of node, or using a standard local installation from the command line. Of course, both can be mixed (some nodes running as containers and other by command line).

Datasets

For now, there are two models for training: MNIST and Chest X-Ray. For using the MNIST one you don't need to install anything else because
the client node downloads the dataset when it runs the training, but for the Chest X-Ray model you'll need a dataset to get it working.

Download the dataset from https://data.mendeley.com/public-files/datasets/rscbjbr9sj/files/f12eaf6d-6023-432f-acc9-80c9d7393433/file_downloaded, and uncompress it wherever you want on the client node machine, in a folder called *chest_xray*. The final structure must be (other content in this folder will be ignored):



By default, the client node looks for it at *GLOBAL_DATASETS/chest_xray*. The variable *GLOBAL_DATASETS* is defined in the configuration file *client/config.py*.

If you're going to run the client node using Docker, you must pass a volume as a container parameter to indicate where you have the datasets:

```
-v /your_datasets_directory:/federated-learning-network/datasets
```

In particular, for Chest X-Ray training, it'll expect a directory *chest_xray* in your dataset's directory with at least two folders *train* and *test* with x-ray images.

Docker installation

Create the Docker image of the server:

```
cd server
docker build -t fl-server -f Dockerfile .
```

Run the server:

```
docker run --rm --name fl-server -p 5000:5000 fl-server:latest
```

This command will delete the server container after stopping it. It runs the server on port 5000.

For the client, the first step is creating the Docker image:

```
cd client
docker build -t fl-client -f Dockerfile .
```

Running the project

Now there can be two different scenarios: running nodes on the same IP address, or running each node on a different IP address.

Bear always in mind than we can choose the ports we want if they are free. The ports used in these examples are just that, examples.

Same machine

If our IP address is for example 192.168.1.20, and we have the server running on port 5000, we can run several Docker clients in different ports:

```
docker run --rm --name fl-client-5001 -p 5001:5000 -e
CLIENT_URL='http://192.168.1.20:5001' -e
SERVER_URL='http://192.168.1.20:5000' -v
/your_datasets_directory:/federated-learning-network/datasets fl-
client:latest

docker run --rm --name fl-client-5002 -p 5002:5000 -e
CLIENT_URL='http://192.168.1.20:5002' -e
SERVER_URL='http://192.168.1.20:5000' -v
/your_datasets_directory:/federated-learning-network/datasets fl-
client:latest

docker run --rm --name fl-client-5003 -p 5003:5000 -e
CLIENT_URL='http://192.168.1.20:5003' -e
SERVER_URL='http://192.168.1.20:5000' -v
/your_datasets_directory:/federated-learning-network/datasets fl-
client:latest

docker run --rm --name fl-client-5004 -p 5004:5000 -e
CLIENT_URL='http://192.168.1.20:5004' -e
SERVER_URL='http://192.168.1.20:5000' -v
```

```
/your_datasets_directory:/federated-learning-network/datasets fl-
client:latest
```

If the server is running on another IP address, simply change the variable *SERVER_URL* accordingly.

IMPORTANT: To be able to use the Chest X-Ray model training follow the instructions of *Training the Chest X-Ray model* section.

Every node on a different IP address
If the IP address of the server is, for instance, at 192.168.1.100, and every client will be running on different IP addresses, we can do:

```
docker run --rm --name fl-client -p 5000:5000 -e
CLIENT_URL='http://192.168.1.28:5000' -e
SERVER_URL='http://192.168.1.100:5000' -v
/your_datasets_directory:/federated-learning-network/datasets fl-
client:latest
```

For other clients, simply use the right IP address of each one:

```
docker run --rm --name fl-client -p 5000:5000 -e
CLIENT_URL='http://192.168.1.50:5000' -e
SERVER_URL='http://192.168.1.100:5000' -v
/your_datasets_directory:/federated-learning-network/datasets fl-
client:latest

docker run --rm --name fl-client -p 5000:5000 -e
CLIENT_URL='http://192.168.1.60:5000' -e
SERVER_URL='http://192.168.1.100:5000' -v
/your_datasets_directory:/federated-learning-network/datasets fl-
client:latest

docker run --rm --name fl-client -p 5000:5000 -e
CLIENT_URL='http://192.168.1.70:5000' -e
SERVER_URL='http://192.168.1.100:5000' -v
/your_datasets_directory:/federated-learning-network/datasets fl-
client:latest

docker run --rm --name fl-client -p 5000:5000 -e
CLIENT_URL='http://192.168.1.80:5000' -e
SERVER_URL='http://192.168.1.100:5000' -v
/your_datasets_directory:/federated-learning-network/datasets fl-
client:latest
```

Command line
If Docker is not an option, then you must install everything and running from the command line. Python version must be 3.8, I haven't tested it with 3.9 or <3.8 versions.

The best way is to have an isolated environment using conda or similar environment managers. If you use miniconda or conda, just do:

```
conda create --name fedlearning python=3.8
conda activate fedlearning
```

Once you're ready to install packages, do this:

```
pip install torch torchvision
pip install tensorflow
pip install fastai
pip install python-dotenv
pip install aiohttp[speedups]
pip install flask
```

Running the project
Central node
That's very simple, just go to *federated-learning-network/server* and execute:

```
flask run
```

It'll start a central node in *http://localhost:5000*. To see that's running well, open a browser and go to that URL. You'll see the dashboard of the network.

Clients
Open a new console, or just do it in another computer which has access to the server.
Go to federated-learning-network/client and execute:

```
export CLIENT_URL='http://localhost:5001'
flask run --port 5001
```

Do that for every client, changing the listening port. You'll see some log traces telling the client has started and has registered in the network:

```
Registering in server: http://127.0.0.1:5000
Doing request http://127.0.0.1:5000/client
Response received from registration: <Response [201]>
Client registered successfully
```

If you refresh the central's node dashboard you can see all the clients registered in the network.

Training sessions
Once we have the central node and clients running properly and registered, just open the dashboard and click on the *Launch training* button. This action will launch a training session between all the clients registered. You can see the progress of the training in each

client's console. For example, for MNIST training you will see something like this in the client node console:

```
Federated Learning config:
--Learning Rate: 1.0
--Epochs: 20
--Batch size: 256

Training started...
Accuracy of model trained at epoch 1 : 0.9118
Accuracy of model trained at epoch 2 : 0.9118
Accuracy of model trained at epoch 3 : 0.9118
Accuracy of model trained at epoch 4 : 0.9118
Accuracy of model trained at epoch 5 : 0.8824
Accuracy of model trained at epoch 6 : 0.8824
Accuracy of model trained at epoch 7 : 0.9118
Accuracy of model trained at epoch 8 : 0.9118
Accuracy of model trained at epoch 9 : 0.9118
Accuracy of model trained at epoch 10 : 0.9118
Accuracy of model trained at epoch 11 : 0.9118
Accuracy of model trained at epoch 12 : 0.9118
Accuracy of model trained at epoch 13 : 0.9118
Accuracy of model trained at epoch 14 : 0.9118
Accuracy of model trained at epoch 15 : 0.9118
Accuracy of model trained at epoch 16 : 0.9412
Accuracy of model trained at epoch 17 : 0.9412
Accuracy of model trained at epoch 18 : 0.9412
Accuracy of model trained at epoch 19 : 0.9412
Accuracy of model trained at epoch 20 : 0.9412
Training finished...
```

You can do more training sessions afterwards and see how the model improves.

Customization
You can change some training parameters (epochs, batch size and learning rate) at:

```
federated-learning-network/server/server.py start_training method
```

In the future it'll be possible to do it from the central node's dashboard.

Known issues
There's no persistence implemented yet, so every time you start servers & clients the model will be initialized with random values and must be trained from the beginning.

This is a very early version, so it has room for lots of improvements, so new features will be added.

Annex II: Simulator installation and user's guide

The simulator needs Python 3.8 and has two dependencies that can be installed using *pip*:

```
pip install --quiet --upgrade tensorflow_federated
pip install --quiet --upgrade nest_asyncio
```

For running the simulation, simply execute the *simulation.py* file:

```
python simulation.py
```

The number of clients, the batch size of the subsets and the total rounds of training can be changed in the line 10 of the file.

```
10. federated_simulator = FederatedSimulator(emnist_train, emnist_test, batch_form
    at, create_keras_model,
11.                                          num_clients=10, batch_size=20, rounds
    =100)
```

Annex III: Simulator source's code

The simulator is made out of three files that must be in the same directory:

simulator.py

```
1.  import tensorflow as tf
2.  import tensorflow_federated as tff
3.  import nest_asyncio
4.  from federated_simulator import FederatedSimulator
5.  from federated_external_model import batch_format, create_keras_model
6.
7.  nest_asyncio.apply()
8.  emnist_train, emnist_test = tff.simulation.datasets.emnist.load_data()
9.
10. federated_simulator = FederatedSimulator(emnist_train, emnist_test, batch_form
    at, create_keras_model,
11.                                           num_clients=10, batch_size=20, rounds
    =100)
12. federated_training_data = federated_simulator.get_federated_training_data()
13.
14.
15. @tf.function
16. def client_update(model, dataset, server_weights, client_optimizer):
17.     """The most important function, It's the training of each client."""
18.     # Initialize client weights with server weights.
19.     client_weights = model.weights.trainable
20.     tf.nest.map_structure(lambda x, y: x.assign(y),
21.                           client_weights, server_weights)
22.
23.     # For each batch in the dataset, compute the gradients using the client op
    timizer
24.     for batch in dataset:
25.         with tf.GradientTape() as tape:
26.             outputs = model.forward_pass(batch)
27.
28.         grads = tape.gradient(outputs.loss, client_weights)
29.         grads_and_weights = zip(grads, client_weights)
30.         client_optimizer.apply_gradients(grads_and_weights)
31.
32.     return client_weights
33.
34.
35. @tf.function
36. def server_update(model, mean_client_weights):
37.     """Updates the server weights with an average of the client wegiths calcul
    ated by each client"""
38.     # Get the model weights
39.     model_weights = model.weights.trainable
40.     # Assign the mean of the clients weights to the server model weights
41.     tf.nest.map_structure(lambda x, y: x.assign(y),
42.                           model_weights, mean_client_weights)
43.     return model_weights
44.
45.
46. def model_fn():
47.     """Creates the Keras model with a loss function, accuray as metric and the
     specification of the input data"""
48.     keras_model = create_keras_model()
49.     return tff.learning.from_keras_model(
50.         keras_model,
51.         input_spec=federated_training_data[0].element_spec,
52.         loss=tf.keras.losses.SparseCategoricalCrossentropy(),
```

```
        metrics=[tf.keras.metrics.SparseCategoricalAccuracy()]
    )


@tff.tf_computation
def server_init():
    """Initialization of the server model"""
    model = model_fn()
    return model.weights.trainable


dummy_model = model_fn()

# Definition of Federated Types for Federated Functions
# The arguments of some of the annotations are special types that define the type of the data and where it's used (server or client).
tf_dataset_type = tff.SequenceType(dummy_model.input_spec)
model_weights_type = server_init.type_signature.result
federated_server_type = tff.FederatedType(model_weights_type, tff.SERVER)
federated_dataset_type = tff.FederatedType(tf_dataset_type, tff.CLIENTS)


# Now come the federated functions annotated with Tensorflow Federated special annotations.
# These functions are used by the framework to run the simulation.
# Each federated function uses the corresponding regular function defined previously.
@tff.federated_computation
def initialize_fn():
    return tff.federated_value(server_init(), tff.SERVER)


@tff.tf_computation(tf_dataset_type, model_weights_type)
def client_update_fn(tf_dataset, server_weights):
    model = model_fn()
    client_optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
    return client_update(model, tf_dataset, server_weights, client_optimizer)


@tff.tf_computation(model_weights_type)
def server_update_fn(mean_client_weights):
    model = model_fn()
    return server_update(model, mean_client_weights)


@tff.federated_computation(federated_server_type, federated_dataset_type)
def next_fn(server_weights, federated_dataset):
    # Broadcast the server weights to the clients.
    server_weights_at_client = tff.federated_broadcast(server_weights)

        # Each client computes their updated weights.
        client_weights = tff.federated_map(
            client_update_fn, (federated_dataset, server_weights_at_client)
        )

        # The server averages these updates.
        mean_client_weights = tff.federated_mean(client_weights)

        # The server updates its model.
        server_weights = tff.federated_map(server_update_fn, mean_client_weights)

        return server_weights

```

```
113.       # The Federated Lerning algorithm is an 'Iterative Process' which first
     initializes the server,
114.       # then run next_fn the number of rounds defined at the beginning of the
     simulation.
115.       federated_algorithm = tff.templates.IterativeProcess(
116.           initialize_fn=initialize_fn,
117.           next_fn=next_fn
118.       )
119.       federated_simulator.run_simulation(federated_algorithm)
120.       federated_simulator.evaluate()
```

federated_simulator.py

```
import numpy as npimport numpy as np
import tensorflow as tf


class FederatedSimulator:
    def __init__(self, clients_training_data, clients_test_data, batch_format_fn, create_keras_model_fn, num_clients=10, batch_size=20, rounds=50):
        np.random.seed(0)
        # TODO arguments of the simulator
        self.num_clients = num_clients
        self.batch_size = batch_size
        self.rounds = rounds
        self.__batch_format_fn = batch_format_fn
        self.__create_keras_model = create_keras_model_fn
        self.__federated_training_data = self.__build_federated_training_data(clients_training_data)
        self.__federated_test_data = self.__preprocess(clients_test_data.create_tf_dataset_from_all_clients())
        self.__server_state = None

    def run_simulation(self, federated_algorithm):
        self.__server_state = federated_algorithm.initialize()
        for round in range(self.rounds):
            self.__server_state = federated_algorithm.next(self.__server_state, self.__federated_training_data)

    def evaluate(self):
        keras_model = self.__create_keras_model()
        keras_model.compile(
            loss=tf.keras.losses.SparseCategoricalCrossentropy(),
            metrics=[tf.keras.metrics.SparseCategoricalAccuracy()]
        )
        keras_model.set_weights(self.__server_state)
        keras_model.evaluate(self.__federated_test_data)

    def get_federated_training_data(self):
        return self.__federated_training_data

    def __preprocess(self, dataset):
        return dataset.batch(self.batch_size).map(self.__batch_format_fn)

    def __build_federated_training_data(self, training_data):
        client_ids = np.random.choice(training_data.client_ids, size=self.num_clients, replace=False)
        federated_training_data = [self.__preprocess(training_data.create_tf_dataset_for_client(x))
                                   for x in client_ids
                                   ]
        return federated_training_data
```

federated_external_model.py

```
1.  import tensorflow as tf
2.
3.
4.  def create_keras_model():
5.      return tf.keras.models.Sequential([
6.          tf.keras.layers.Input(shape=(784,)),
7.          tf.keras.layers.Dense(10, kernel_initializer='zeros'),
8.          tf.keras.layers.Softmax(),
9.      ])
10.
11.
12. def batch_format(element):
13.     return (tf.reshape(element['pixels'], [-1, 784]),
14.             tf.reshape(element['label'], [-1, 1]))
```