



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA



UNIVERSITAT POLITÈCNICA DE VALÈNCIA

School of Informatics

Developing a framework for SPADE agents management  
that implement consensus learning algorithms.

Master's Thesis

Master's Degree in Informatics Engineering

AUTHOR: Matagne , Miro-Manuel

Tutor: Carrascosa Casamayor, Carlos

Cotutor: Rincón Arango, Jaime Andrés

ACADEMIC YEAR: 2022/2023



# Acknowledgements

---

This thesis is the fruit of more than 8 months of work in collaboration with my supervisor Pr. Carlos Carrascosa Casamayor, and my co-supervisor Pr. Jaime Andrés Rincón Arango, both professors at the Universitat Politècnica de València (UPV). I would like to thank them both for this opportunity to join their team and to allow me to help them out in a research subject that I am passionate about. I would also like to thank them for all the support they provided me throughout this work, and all the time they dedicated to help me reach the final objectives. The confidence and trust they had in me, and the respect they showed towards my work are things that I am very grateful about.

I would also like to thank Aaron Pico Pascual and Francisco Engiux Andres, 2 students at the faculty of Informatics at the UPV, who were also producing their Master Thesis about related subjects, and who helped out with several tasks and the understanding of certain concepts.

I am also very grateful for the help that Pr. Miguel Rebollo Pedruelo has given me throughout this project, especially on the theoretical point of view.

It was a huge honor for me to be able to write an academical paper, along with Pr. Carlos Carrascosa Casamayor, Pr. Jaime Andrés Rincón Arango, Pr. Miguel Rebollo Pedruelo and Aaron Pico Pascual, which was written for the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS) [12].

Finally, this thesis marks the end of my stay of a year and a half as a Double Master student at Universitat Politècnica de València. I am very grateful to my home university (Université Libre de Bruxelles) for this opportunity, as well as to UPV for welcoming me and providing me with such a rewarding professional and personal experience.

# Abstract

The rapid pace of development of Artificial Intelligence, which has undeniably become a key tool in our society and in industries, has led to the development of new Machine Learning techniques. Federated Learning improves the performance of the training of a Machine Learning model by distributing this process across multiple clients and combining the models in a central server. Each client trains a model on a subset of the full dataset, and sends its model to the central server upon completion of a round of training. Although Federated Learning is already heavily used and provides good results, the main drawback is that the central server receives all models from all clients, which constitutes a single point of failure as well as a possible bottleneck. This led to the apparition of Decentralized Federated Learning architectures, where the agents communicate between themselves and no central server is involved. However, it was noticed that this architecture could be improved further in order to reduce the idle time of all clients; which leads to the development of a new Federated Learning algorithm presented in this paper : *Asynchronous Decentralized Federated Learning*. The key specificity of this solution is that the idle time of all clients is removed, and the model training is therefore performed quicker. Furthermore, this study relates this algorithm to the concept of Multi-Agent Systems, which correspond to systems of agents cooperating to achieve common goals. In this sense, this work presents a concrete implementation of the Asynchronous Decentralized Federated Learning algorithm in a standalone and easy to use application that was built upon the SPADE Python library, which allows to manage Multi-Agent Systems. This application is destined to be a concrete product that users can easily use in order to launch, control, monitor and supervise the execution of an Asynchronous Decentralized Federated Learning running on multiple clients. It is shown experimentally in this work that the developed algorithm is indeed more efficient than the synchronous algorithms in certain use cases, although some ideas can be identified as future improvements. Furthermore, the developed application meets its objectives, it is to this day accessible in an easy way by any user, and can be used at a large scale.

**Key words:** Machine Learning, Federated Learning, Multi-Agent Systems, Consensus, Asynchronous Decentralized Federated Learning

---

---

# Résumé

Le développement rapide de l'Intelligence Artificielle, qui est indéniablement devenue un outil clé dans notre société et dans l'industrie, a conduit au développement de nouvelles techniques d'Apprentissage Automatique (ou *Machine Learning*). L'Apprentissage Fédéré (ou *Federated Learning*) améliore les performances de la phase d'entraînement d'un modèle d'Apprentissage Automatique en répartissant ce processus entre plusieurs clients et en combinant ces modèles au sein d'un serveur central. Chaque client entraîne un modèle sur un sous-ensemble de l'ensemble des données disponibles et envoie son modèle au serveur central à la fin d'un cycle d'entraînement. Bien que l'Apprentissage Fédéré soit déjà largement utilisé et fournisse de bons résultats, le principal inconvénient est que le serveur central reçoit tous les modèles de tous les clients, ce qui constitue un point de défaillance unique ainsi qu'une éventuelle limitation en terme de performances. Ceci a été l'élément déclencheur pour l'apparition d'architectures d'Apprentissage Fédéré décentralisées, où les agents communiquent entre eux et aucun serveur central n'est requis. Cependant, il a été remarqué que cette architecture pouvait être encore améliorée afin de réduire le temps d'inactivité de tous les clients; ce qui a conduit au développement d'un nouvel algorithme d'apprentissage fédéré présenté dans cet article : l'Apprentissage Fédéré Décentralisé Asynchrone. La principale spécificité de cette solution est que les clients ne sont plus jamais inactifs, et l'apprentissage du modèle est donc effectué plus rapidement. De plus, cette étude relie cet algorithme au concept de Systèmes Multi-Agents, qui correspondent à des systèmes d'agents coopérant dans le but d'atteindre des objectifs en commun. Ce travail présente une implémentation concrète de l'algorithme d'Apprentissage Fédéré Décentralisé Asynchrone au travers du développement d'une application facile à utiliser basée sur la librairie SPADE du langage Python, qui permet de gérer des Systèmes Multi-Agents. L'objectif est que cette application soit un produit concret que les utilisateurs puissent facilement utiliser pour lancer, contrôler, surveiller et superviser l'exécution de l'algorithme de l'Apprentissage Fédéré Décentralisé Asynchrone sur plusieurs clients. Il est démontré expérimentalement dans ce travail que l'algorithme développé est effectivement plus efficace que les algorithmes synchrones dans certains cas d'utilisation, bien que certaines améliorations futures puissent être identifiées. De plus, l'application développée remplit ses objectifs, elle est à ce jour accessible de manière simple par tout utilisateur, et peut être utilisée à grande échelle.

**Mots clé:** Apprentissage automatique, Apprentissage Fédéré, Systèmes Multi-Agents, Consensus, Apprentissage Fédéré Décentralisé Asynchrone

---

# Resumen

El desarrollo rápido de la Inteligencia Artificial, que se ha convertido indiscutiblemente en una herramienta clave en nuestra sociedad y en las industrias, ha dado lugar al desarrollo de nuevas técnicas de Aprendizaje Automático (o *Machine Learning*). El Aprendizaje Federado (o *Federated Learning*) mejora el rendimiento del entrenamiento de un modelo de Aprendizaje Automático distribuyendo este proceso entre varios clientes y combinando estos modelos en un servidor central. Cada cliente entrena un modelo con subconjunto del conjunto de datos completo, y envía su modelo al servidor central al finalizar una ronda de entrenamiento. Aunque el Aprendizaje Federado ya se utiliza mucho y proporciona buenos resultados, el principal inconveniente es que el servidor central recibe todos los modelos de todos los clientes, lo que constituye un único punto de fallo, y también un posible cuello de botella. Esto dio lugar a la aparición de las arquitecturas de Aprendizaje Federado Descentralizado, en las que los agentes comunican entre sí y no interviene ningún servidor central. Sin embargo, se observó que esta arquitectura podría mejorarse para reducir el tiempo de inactividad de todos los clientes; lo que dio lugar al desarrollo de un nuevo algoritmo de Aprendizaje Federado presentado en este artículo: el *Aprendizaje Federado Descentralizado Asíncrono*. La especificidad clave de esta solución es que se elimina el tiempo de inactividad de todos los clientes y, por tanto, el entrenamiento del modelo se realiza más rápidamente. Además, este estudio relaciona este algoritmo con el concepto de Sistemas Multiagente, que corresponden a sistemas de agentes que cooperan para alcanzar objetivos comunes. En este sentido, este trabajo presenta una implementación concreta del algoritmo de Aprendizaje Federado Descentralizado Asíncrono en una aplicación independiente y fácil de usar que fue construida sobre la librería SPADE del lenguaje Python, que permite gestionar Sistemas Multiagente. Esta aplicación está destinada a ser un producto concreto que los usuarios puedan utilizar fácilmente para lanzar, controlar, monitorizar y supervisar la ejecución del algoritmo de Aprendizaje Federado Descentralizado Asíncrono ejecutado en múltiples clientes. Se demuestra experimentalmente en este trabajo que el algoritmo desarrollado es más eficiente que los algoritmos síncronos en ciertos casos de uso, aunque se pueden identificar algunas ideas como futuras mejoras. Además, la aplicación desarrollada cumple sus objetivos, es al día de hoy accesible de forma sencilla por cualquier usuario, y puede ser utilizada a gran escala.

**Palabras clave:** Aprendizaje Automático, Aprendizaje Federado, Sistemas Multiagentes, Consensus, Aprendizaje Federado Descentralizado Asíncrono

---

# Contents

---

<b>Contents</b>	<b>vii</b>
<b>List of Figures</b>	<b>ix</b>
<b>List of Tables</b>	<b>xi</b>

---

<b>1 Introduction</b>	<b>3</b>
1.1 Motivation	3
1.2 Objectives	4
1.3 Desired Impact	5
1.4 Structure	5
<b>2 State of the art</b>	<b>7</b>
2.1 Multi-Agent Systems	7
2.1.1 Agents	8
2.1.2 Environments	9
2.1.3 Inter-Agent Communication	9
2.1.4 SPADE	11
2.2 Federated Learning	13
2.2.1 Background on Artificial Intelligence and Machine Learning	13
2.2.2 Basic principles of Federated Learning	16
2.2.3 Centralized and Decentralized Federated Learning	19
2.2.4 Asynchronous Centralized Federated Learning	25
2.3 Consensus-Based Learning	27
2.3.1 Theory of Consensus	27
2.4 Proposition	30
2.5 Starting point of the implementation	30
2.6 Preliminary analysis	31
2.6.1 Definition of requirements	31
2.6.2 Risk analysis	32
<b>3 Elaboration of the solution</b>	<b>35</b>
3.1 Work Plan	35
3.2 Creating the Asynchronous Decentralized Federated Learning algorithm	35
3.2.1 Asynchronous Decentralized Federated Learning scheme	35
3.2.2 Asynchronous consensus	40
3.2.3 Algorithm	41
3.3 Datasets	43
3.3.1 MNIST	43
3.3.2 Fashion MNIST	43
3.4 Models	44
3.4.1 Multilayer Perceptrons	45
3.4.2 Convolutional Neural Networks	46
3.5 Model Training	50
3.6 Consensus	50
3.7 Agents	50
3.7.1 Agent architecture	51

3.7.2	Agent Web Interface . . . . .	56
3.7.3	Launcher Agent . . . . .	57
3.8	Logging . . . . .	59
3.9	Code structure . . . . .	60
3.10	Containerizing the solution . . . . .	62
3.10.1	Docker . . . . .	62
3.10.2	Creating a Docker image of the program . . . . .	63
<b>4</b>	<b>Experimental results</b>	<b>67</b>
4.1	Evaluation of the solution . . . . .	67
4.2	Comparison with a single training agent . . . . .	68
4.3	Comparison with Synchronous Decentralized Federated Learning . . . . .	69
4.3.1	Influence of the number of agents . . . . .	72
<b>5</b>	<b>Limitations and bottlenecks</b>	<b>75</b>
5.1	Length of the messages . . . . .	75
5.2	Number of agents running on the same machine . . . . .	78
5.3	XMPP server . . . . .	79
<b>6</b>	<b>Methodology</b>	<b>81</b>
6.1	Tools . . . . .	81
6.1.1	Notion . . . . .	81
6.1.2	PyCharm . . . . .	81
6.1.3	Git . . . . .	82
6.1.4	Zotero . . . . .	82
6.1.5	Overleaf . . . . .	83
6.2	Project Management . . . . .	83
<b>7</b>	<b>Further improvements</b>	<b>85</b>
7.1	Improving the current solution . . . . .	85
7.1.1	Limiting the length of the messages . . . . .	85
7.1.2	Launch all agents at once . . . . .	86
7.2	Future works . . . . .	86
<b>8</b>	<b>Conclusion</b>	<b>89</b>
8.1	Synthesis . . . . .	89
8.2	Relation with the pursued studies . . . . .	90
	<b>Bibliography</b>	<b>91</b>
<b>A</b>	<b>Log files</b>	<b>97</b>
A.1	Message log file . . . . .	97
A.2	Training log file . . . . .	98
A.3	Weight log file . . . . .	99
A.4	Epsilon log file . . . . .	100
A.5	Training time log file . . . . .	101
<hr/>		
Appendices		
<b>B</b>	<b>Link with Sustainable Development Goals (SDG)</b>	<b>103</b>
B.1	Sustainable Development Goals . . . . .	104
B.2	Relating this study to SGDs . . . . .	104
B.2.1	Artificial Intelligence and Sustainable Development Goals . . . . .	104
B.2.2	Federated Learning and Sustainable Development Goals . . . . .	106
B.2.3	Dangers of AI . . . . .	106



# List of Figures

---

2.1	Example of an agent network graph used in a MAS . . . . .	8
2.2	Example of the usage of XMPP protocol . . . . .	10
2.3	Example of a SPADE agent’s default graphical interface . . . . .	12
2.4	Example of a SPADE agent behaviour’s default graphical interface . . . . .	13
2.5	Steps of a Federated Learning training round . . . . .	18
2.6	Usage of Federated Learning in Gboard. This picture was taken from [46]	19
2.7	Graph representing the agents and their connections . . . . .	20
2.8	Steps of a Synchronous Decentralized Federated Learning training round	23
2.9	Comparison between a centralized architecture and various implementa- tions of asynchronous architectures. This figure was taken from [38]. . . . .	24
2.10	Results of experiments on the MNIST dataset using the Asynchronous De- centralized Federated Learning algorithm described in [15]. This figure was taken from [15]. . . . .	25
2.11	Comparison between a synchronous and asynchronous implementation of Centralized Federated Learning. This figure was taken from [14]. . . . .	26
2.12	Results of experiments on the following datasets : FirRec, Air Quality, Ex- traSensory, Fashion-MNIST, with several Federated Learning algorithms. This figure was taken from [14]. . . . .	26
3.1	Steps of an Asynchronous Decentralized Federated Learning training round	39
3.2	Example of MNIST dataset samples . . . . .	43
3.3	The 10 Fashion MNIST classes and examples of dataset samples . . . . .	44
3.4	Multilayer Perceptron architecture . . . . .	46
3.5	Convolutional Neural Network architecture . . . . .	47
3.6	Example of 3 convolution operations. This figure was taken from [72] . . . . .	48
3.7	Example of a max pooling with a $2 \times 2$ kernel. This figure was taken from [72] . . . . .	49
3.8	First proposed solution . . . . .	54
3.9	Second proposed solution . . . . .	55
3.10	Third proposed solution . . . . .	55
3.11	Agent’s graphical interface . . . . .	56
3.12	Graphical Interface when the user selects the option to create the agent by uploading a Network Graph . . . . .	58
3.13	Graphical Interface when the user selects the option to create the agent without a Network Graph . . . . .	59
3.14	How Docker allows to create containerized applications. This figure was taken from <a href="https://www.docker.com/resources/what-container/">https://www.docker.com/resources/what-container/</a> . . . . .	62
4.1	Evolution of training and test accuracy in function of the number of itera- tions, both for a Multi-Layer Perceptron (a) and for a Convolutional Neural Network (b) . . . . .	68
4.2	Evolution of one MLP weight of 2 agents throughout 20 rounds of training	68

4.3	Evolution of training and test accuracy for 2 agents applying the ADFL algorithm and one single agent training locally in function of the number of iterations, both for a Multi-Layer Perceptron (a) and for a Convolutional Neural Network (b) . . . . .	69
4.4	Graph representing the network of agents used to conduct experiments to compare the ADFL setting to the SDFL setting . . . . .	70
4.5	Evolution of test accuracy round by round for the synchronous and asynchronous setting . . . . .	70
4.6	Evolution of test accuracy in function of time for the synchronous and asynchronous setting . . . . .	71
4.7	Evolution of average test accuracy in function of the number of rounds for a system with 2 agents and a system with 7 agents . . . . .	72
5.1	Evolution of test accuracy in function of the number of neurons in the hidden layer of the used MLP . . . . .	77
6.1	Example of a team meeting summary on Notion . . . . .	82
7.1	Evolution of normalized and unnormalized Mutual Information in function of the number of users in the Federated Learning setting, with 3 different models. This figure was taken from [18]. . . . .	87
A.1	Example of an message log file . . . . .	97
A.2	Example of a training log file . . . . .	98
A.3	Example of a weight log file . . . . .	99
A.4	Example of an epsilon log file . . . . .	100
A.5	Example of an training time log file . . . . .	101

# List of Tables

---

2.1	Requirements for the development of the ADFL algorithm . . . . .	31
2.2	Requirements for the development of the application implementing the ADFL algorithm . . . . .	32
2.3	Risks involved in the development of the application and measures to mitigate them . . . . .	33
3.1	Work plan. Each phase is described and is attributed a certain number of working hours as well as a difficulty. . . . .	36
3.2	Characteristics of the MAS environment implemented in the solution . . . . .	51
5.1	Length of the sent messages in function of the $x$ parameter of the MLP . . . . .	75
5.2	Average, minimum and maximum times taken to send a message in function of the length of this message . . . . .	76
5.3	Duration of the MLP model training on the MNIST dataset in function of the size of the model . . . . .	77
5.4	Duration of the MLP model training on the MNIST dataset in function of the number of agents training on the same device . . . . .	78



# Glossary

---

- ADFL** Asynchronous Decentralized Federated Learning. 4, 30, 31, 36, 41, 69, 73
- AI** Artificial Intelligence. 3, 13, 14, 18
- CNN** Convolutional Neural Network. 46–49, 58, 67, 69, 86
- DAI** Distributed Artificial Intelligence. 7
- FL** Federated Learning. 3, 4, 6, 16, 18, 23, 25, 79
- IoT** Internet of Things. 7, 8
- MAS** Multi-Agent Systems. 3, 4, 6–13
- ML** Machine Learning. 3, 13, 14, 18, 78
- MLP** Multilayer Perceptron. 45, 49, 58, 67, 69, 70, 75–78, 86
- NN** Neural Network. 45
- SDFL** Synchronous Decentralized Federated Learning. 4, 20, 23, 36, 72
- SDG** Sustainable Development Goals. 6
- SPADE** Smart Python Agent Development Environment. 3–5, 11, 12, 36, 51–53, 57
- XMPP** Extensible Messaging and Presence Protocol. 9–12, 79, 85



---

---

# CHAPTER 1

## Introduction

---

The code related to this project is available publicly at the following link :

<https://github.com/miromatagne/spade-adfl>

### 1.1 Motivation

---

Artificial Intelligence (AI) is today widely accepted as a society-changing technology. Its impact on a wide range of industries such as transportation [1], healthcare [5], banking [32], real estate [30] and many more is considerable. Researchers are constantly searching for ways to improve existing Artificial Intelligence solutions or to create new ones in order to respond to specific needs in our society, and to adapt to other technology related trends such as Internet of Things (IoT) or cloud computing. This paper focuses more specifically into Machine Learning (ML), which is a branch of Artificial Intelligence which "gives computers the ability to learn without explicitly being programmed" [61]. In other terms, ML allows machines to learn from past data without being explicitly programmed and produce predictions. The main component of ML is called a model, which is the program that allows to learn from data and make predictions.

In 2017, Google presented the idea of Federated Learning (FL) [45, 46], which is a new form of Machine Learning involving multiple devices, called clients, that cooperate together. The key idea is that multiple devices train a Machine Learning model and then share their knowledge in a central server that aggregates all of their results.

This aggregation of all the models by the central server is a very important step, and makes use of a concept called *consensus*. This concept is the subject of a lot of research, in order to be able to prove that the way the central server combines the models is efficient and leads to appropriate results. Algorithms making use of consensus in the ML domain can be referred to as *Consensus-based Learning* algorithms.

In parallel, the last decades have seen the apparition and development of Multi-Agent Systems (MAS). These are systems that involve multiple *agents* that cooperate in order to achieve a particular goal [16]. The particularity of these systems is that they are decentralized, the agents can communicate between themselves without needing the presence of a central server. In practise, frameworks and libraries have been created in order to facilitate the creation of Multi-Agent Systems, such as SPADE, a Python library developed

by Pr. Javi Palanca [53], professor at Universitat Politècnica de València (UPV).

Due to their collaborative nature, it is understandable that Federated Learning and MAS could be grouped in a single solution, where the different agents of the MAS play the role of the different clients of FL. Furthermore, the Federated Learning setting could possibly benefit from the decentralized aspect of Multi-Agent Systems. Indeed, we could imagine a version of FL where the centralized server is absent, and where the various clients communicate directly between themselves in order to exchange information. This, of course, presents some limitations and requires to adapt the original FL algorithm to this new environment.

A first solution to combining the principles of FL and MAS is called *Synchronous Decentralized Federated Learning (SDFL)*, which, in broad terms, consists in a solution where agents share the information about the model they are training to other neighbours in a synchronous fashion. Generally speaking, this means that until all agents have not sent their results, the next round of this iterative process can not begin.

An alternative solution which works as the SDFL solution but in an asynchronous fashion, which we will call *Asynchronous Decentralized Federated Learning (ADFL)*, could avoid all these waiting times and improve the SDFL algorithm. This is a new algorithm that will be developed throughout this work, and that will be tested and compared to existing alternatives. In broad terms, the main idea of this new algorithm is that the agents do not need to wait for all the other agents to send their model information before beginning the next iteration, and therefore gain in efficiency.

## 1.2 Objectives

---

The objectives of this work all include the combination of MAS and FL, making use of the SPADE library in order to produce a finished product that is easily usable to run and test these new ML algorithms.

First of all, the first objective is to develop a solution implementing Synchronous Decentralized Federated Learning using SPADE. Experiments should be made in order to evaluate the performance of this solution, and check that it works as desired.

Next, the Asynchronous Decentralized Federated Learning algorithm and specifications should be elaborated. This is a phase where a lot of research and investigation is necessary, in collaboration with the supervisors as well as Pr. Miguel Rebello Pedruelo. There was a lot of theoretical issues to discuss, as well as practical implementation choices. Since this is a new algorithm that had not been created or tested by any other researcher, to our knowledge, a lot of tests and simulations were required to prove the solution would give good results.

Next, once the theoretical algorithm was created, a new solution implementing Asynchronous Decentralized Federated Learning using SPADE must be implemented, in the form of a standalone application that can be easily executed by a user in order to perform ADFL on several devices. This solution should be a easy to execute application, taking advantage of the benefits of using SPADE to provide some abstraction. It is then very im-



portant to compare this solution to the SDFL solution in order to determine if it presents any significant advantages or disadvantages, and to understand in which situation one algorithm could be more performant than another. Furthermore, the interaction with the user must be taken into consideration. Indeed, the user needs to be able to follow the evolution of the execution in real time and monitor how the system is evolving. The user must also be able to enter custom preferences very easily, and the solution should adapt to it and offer a clear and easy to use interface to the user.

Next, an academic paper was written, in collaboration with Pr. Carlos Carrascosa Casamayor, Pr. Jaime Andrés Rincón Arango, Pr. Miguel Rebello Pedruelo and Aaron Pico Pascual. This paper was titled *Asynchronous Consensus for Multi-Agent Systems and its Application to Federated Learning* [12], and was submitted to the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS)<sup>1</sup> which takes place between the 29th of May 2023 and the 2nd of June 2023 in London. A part of this paper was dedicated to the development of a SPADE framework implementing the Asynchronous Decentralized Federated Learning algorithm that was created.

Finally, the last objective of this work is to understand in which circumstances any of these solutions would be of an appropriate use, and understanding what are the limitations or bottlenecks in these architectures. Furthermore, we should think about how the developed solutions could be improved in the future to produce solutions that are more efficient.

### 1.3 Desired Impact

---

First of all, on the research part of this work, a desired impact is to conclude that the developed Asynchronous Decentralized Federated Learning setting presents some advantages compared to other Federated Learning implementations in some use cases. In particular, an experimental comparison with the Synchronous Decentralized Federated Learning setting should be conducted, in the hopes of observing an improvement provided by the newly created algorithm. We hope that, through the publication of the academic paper that was written, this algorithm and its performances could lead to more research in this field and inspire researchers to further improve the solution.

Next, on the more practical and product-oriented part of this work, a desired impact is that the developed application which runs the Asynchronous Decentralized Federated Learning algorithm is performant, easy to use, error-free and intuitive. The application should enable users to execute this algorithm in a very simple manner, which would allow them to easily set up a network of communicating agents implementing the ADFL algorithm.

### 1.4 Structure

---

This work will first of all analyze the state of the art in the domains that are relevant to the study. We first of all explain the principles of Multi-Agent Systems, Federated Learning

---

<sup>1</sup><https://aamas2023.soton.ac.uk/>

and Consensus-Based Learning in order to give the reader all the necessary theoretical background before explaining the developed solution in more detail. Next, an analysis of the state of the art advances in FL systems involving MAS principles in a decentralized fashion will be analysed, and the work that was already carried out in this project before the start of this work will also be detailed.

Next, the elaboration of the produced solution will be explained, before analyzing the results returned by the various experiments made on this solution.

The limitations and bottlenecks that limit the performance of the solution will then be analyzed, in order to explain the experimental results.

Next, the methodology, tools and project management techniques used throughout the project will be detailed.

Finally, some further improvement ideas will be given in order to inform the reader on how these kinds of implementations could possibly be improved in the future, before giving a final conclusion about the work.

In addition, some information will be given in the appendices of this work. Appendix **A** will give examples of some log files that were useful during the elaboration of the solution. Appendix **B** will then present how this work relates to the Sustainable Development Goals (SDG) published by the United Nations (UN).

---

---

## CHAPTER 2

# State of the art

---

Before diving into the implementation of the final solution, a lot of background work and research was necessary. Indeed, this section focuses on giving the required prerequisites and explains the state of the art concepts concerning Multi-Agent Systems, Federated Learning and Consensus-Based Learning.

Then, a deeper analysis of the current state of the art of Federated Learning algorithms is given, analysing thoroughly the latest advances made in this field, as it is the core of this work. In particular, we will focus on the implementations and current advances in Synchronous Decentralized Federated Learning and some research in asynchronous implementations.

Finally, a part of the implementation had already been started before the start of this work by Pr. Carlos Carrascosa Casamayor and Pr. Jaime Andrés Rincón Arango. This previous work will be detailed in this section too, in order to better understand the scope of the implementation.

## 2.1 Multi-Agent Systems

---

Multi-Agent Systems (MAS) is one of multiple algorithms that classify as Distributed Artificial Intelligence (DAI) algorithms. DAI is a class of technologies concerned with the cooperative solution of problems by a decentralized group of agents [28]. This family of algorithms has received a lot of attention in the past years as it allows to resolve efficiently problems that are very difficult to solve on a single machine.

A MAS is a collection of autonomous entities, known as agents, that are capable of learning, making autonomous decisions and collaboratively solve tasks [16]. Indeed, agents are autonomous in the sense that they are capable of executing actions by themselves, but are also collaborative (or social) in the sense that they communicate with each other in order to reach their objective.

These agents can be of many different types, they can be a software, a hardware component, or a combination of both (for example a robot). This is one of the reasons why MAS is often associated with IoT (Internet of Things) [63]. IoT is defined in [23] as a "dynamic global network infrastructure with self-configuration and interoperable communication". In broad terms, IoT refers to a system of *things* such as sensors or software

that can communicate with each other over the Internet. Therefore, to make a parallel between MAS and IoT, MAS agents could be considered as any type of *thing* used in IoT (such as a thermostat, or a drone for instance) [63].

The network of all agents is represented by an agent network graph, which is an undirected graph where each node represents an agent and each edge connecting 2 agents indicates that these 2 agents can communicate one with another. Indeed, not all agents can communicate with each other. An example of an agent network graph is shown in figure 2.1. From this graph, we can see that agent 1 can communicate with agent 2, agent 2 can communicate with agents 1, 3 and 4, etc...

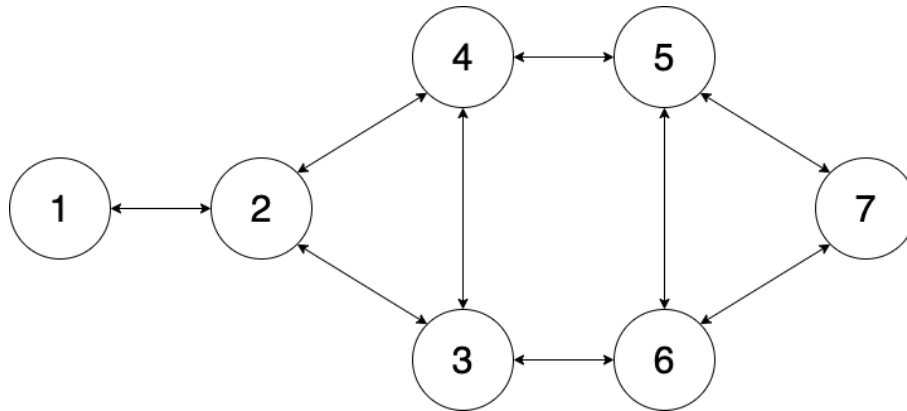


Figure 2.1: Example of an agent network graph used in a MAS

### 2.1.1. Agents

An agent in a MAS is an entity placed in an environment which he can interact with, and that is capable of performing some flexible and autonomous actions inside this environment in order to achieve its goal [73]. The environment designs the *place* where the agent is located. It provides information that the agent can use in order to learn, make decisions and ultimately meet the design objectives. The agent senses information from the environment, which are called parameters. These parameters allow the agent to build up knowledge about the environment. He can then, based on its previous knowledge and the parameters that he sensed, perform an action that affects the environment [73]. Of course, the main characteristic of the MAS is that agents can communicate between themselves, and therefore use knowledge of other agents as well when making the decision of which action to perform.

Agents can be characterized by different attributes that must comply with :

- **Reactivity** : agents must be able to react to changes in the environment. The agent must at all times keep track of the environment's evolution and adapt to the changes in it.
- **Proactivity** : agents must be focused on achieving their objectives, and should therefore take initiatives and the actions should not only be event-driven.
- **Autonomy** : each agent should be able to decide which action to execute autonomously.
- **Sociability** : capacity to communicate with other agents through mechanisms of cooperation (share the knowledge between agents in order to help each other achieve

their objectives), coordination (manage the interdependencies between the activities of the different agents) and negotiation (reach a common agreement about an issue of common interest).

### 2.1.2. Environments

The environment is a key part of a MAS, as it possesses many features that heavily influence the complexity of a MAS, such as :

- **Accessibility** : defines how accurately an agent can obtain data from the environment.
- **Determinism** : refers to the predictability of the consequences of an action on the environment of the MAS.
- **Dynamism** : indicates if there can be some changes in the environment that are not caused by an agent's actions.
- **Continuity** : indicates if the environment is continuous or discrete. In a continuous environment, the agent's state is modified by the environment following a continuous function, whereas in a discrete environment the environment can only place the agent in a finite amount of states.

### 2.1.3. Inter-Agent Communication

Once agents are setup, some mechanisms still need to be put into place in order for them to be able to communicate (communication channels, protocols, ...).

#### Message sending and receiving

In order for the agents to be able to exchange messages and communicate one with another, a specific messaging protocol must be defined. One of the multiple existing protocols, and the one used in this study, is XMPP<sup>1</sup> (Extensible Messaging and Presence Protocol) [60, 70]. It is a set of technologies used for instant messaging, managing presence mechanisms and multiple other routing or messaging purposes. XMPP was built in the Jabber<sup>2</sup> (an instant-messaging protocol) open-source community, in the hopes to improve the existing solution. XMPP offers open and standard protocols, as well as a decentralized alternative to closed instant messaging protocols. Indeed, anyone can run their own XMPP server and handle its own communications. An XMPP server is a server that provides routing services to the agents. Indeed, each agent is identified by its Jabber Identifier (JID), which contains a username and the name of the XMPP server, separated by the symbol @. The XMPP server corresponding to the agent is responsible of sending and receiving the messages that need to be transmitted to other agents. When an agent wants to send a message to another agent, it sends this message to its XMPP server. The XMPP server then finds a way to send this message to the XMPP server of the receiving agent, which finally sends the message to the receiving agent.

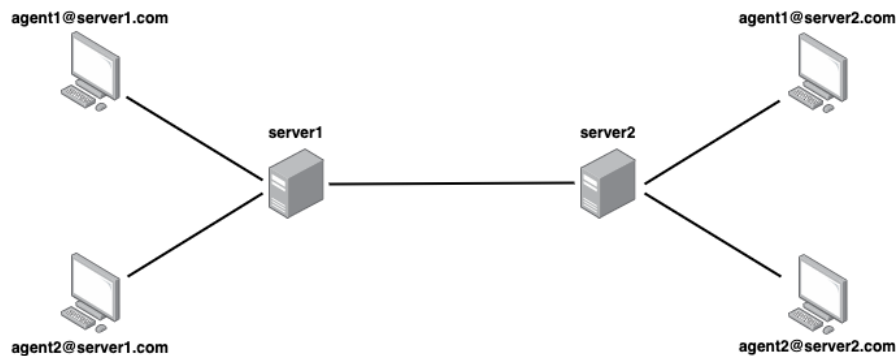
An example is presented at figure 2.2. If agent1@server1.com wants to send a message to agent2@server2.com, he first sends this message to its XMPP server, which is

---

<sup>1</sup>[www.XMPP.org](http://www.XMPP.org)

<sup>2</sup>[www.jabber.com](http://www.jabber.com)

server1. Then, server1 will look for the XMPP server corresponding to agent2@server2.com. He identifies server2 and transfers him the message. server2 then forwards the message to agent2@server2.com.



**Figure 2.2:** Example of the usage of XMPP protocol

XMPP is an extensible protocol, which means that new features (extensions, called XEP) can be easily added to the existing solution. XMPP is nowadays considered as the universal standard protocol for instant messaging by the IETF<sup>3</sup> (Internet Engineering Task Force), and it is widely in the industry. For instance, companies like WhatsApp<sup>4</sup> or Zoom<sup>5</sup> make use of this protocol [69, 17].

### Presence mechanism

An XMPP server does not only handle the routing of the messages as depicted previously. Indeed, it also handles the contact lists of each agent. The contact list of an agent corresponds to a list of all the agents the agent has *subscribed* to, or in clear all the agents that the agent wants to communicate with. The contact list of each agent is stored on the corresponding XMPP server. When an XMPP agent connects (or disconnects) to its XMPP server, the server automatically notifies all the agents on its contact list about this connection (or disconnection). This is therefore a very useful mechanism, since an agent is immediately notified when he can start (or should stop) communicating with an agent he has subscribed to.

Furthermore, an agent can possess a presence state, which is simply an indication about what the agent is doing. For instance, presence states can be Do Not Disturb (when the agent is busy and should not be messaged), or Chat when the agent is waiting to start a conversation.

XMPP also provides a way to manage availability, which is quite similar to the presence states. Indeed, each agent can either be in an available state or an unavailable state. The factors that influence the agent to be in either one of those states is determined by the developer of the agents, and depends on the usage that is made of the MAS. When a previously available agent becomes unavailable, all the agents on its contact list are notified by this event. Similarly, when a previously unavailable agent becomes available,

<sup>3</sup><https://www.ietf.org/>

<sup>4</sup>[www.whatsapp.com](http://www.whatsapp.com)

<sup>5</sup>[www.zoom.us](http://www.zoom.us)

all of the agents on its contact list are notified.

#### 2.1.4. SPADE

SPADE (Smart Python Agent Development Environment) is a multi-agent systems platform written in Python and based on instant messaging using the XMPP protocol [53]. This library is very much adapted to the goal of this research, as it allows to embody Multi-Agent Systems in a simple way, and provides a level of abstraction for Inter-Agent Communication.

Since SPADE is based on XMPP, an XMPP server is necessary in order for SPADE agents to be able to run and communicate between themselves. In the case of this study, a private XMPP server will be used, but multiple online XMPP servers exist that could also be used. When a SPADE agent wants to connect to the server, it needs to have a JID (*username@server*) and a password.

Once the agent is created and connected to its server, it can create one or several behaviours. A behaviour produces a "particular execution pattern designed to support a typical execution requirement of agents in a Multi-Agent System" [53]. In other words, a behaviour represents a task that an agent is expected to carry on during its execution. There are 5 different pre-defined behaviours in SPADE :

- Cyclic Behaviour : performs a task repeatedly, until the behaviour is stopped.
- Periodic Behaviour : performs a task periodically, with a period specified by the user.
- One-Shot Behaviour : performs a task only once as soon as the behaviour is created.
- Timeout Behaviour : performs a task only once, but after a certain time which is specified by the user.
- Finite State Machine Behaviour : performs more complex tasks. This behaviour contains multiple states that are created by the user and allow events to trigger the state switches. All the permitted transitions between the states should also be specified.

All SPADE agents possess a message dispatcher, whose role is to redirect the incoming messages to the agent's behaviour that is expecting that type of message, and to send any message to the XMPP server.

SPADE makes an extensive use of asynchronous programming. Indeed, all behaviours of an agent run simultaneously and execute in parallel, and SPADE tackles this situation by making an extensive use of the Python library AsyncIO, which is used to write concurrent code. In the case of this study, the agent must be able to train its model, but also be ready to send or receive messages simultaneously.

SPADE is a useful library as it addresses issues in current MAS platforms. First of all, it offers a simple, clear and standardized alternative to other solutions. Indeed, SPADE provides an easy to use abstraction to instant messaging, and makes use of XMPP which

is a widely used protocol, in opposition to some other MAS platforms that may have their own specific communication protocols. SPADE also tackles a problem of scalability, as it allows to dynamically allocate communication resources, such as the agents do not suffer when the amount of messages arriving to a server is higher than usual. Furthermore, SPADE allows to connect agents and humans, which is a key aspect of Ambient Intelligence (AmI), a very relevant subject these days. Indeed, since agents use the XMPP protocol, they can easily connect with other humans or even other applications (like Telegram bots for instance [65]). SPADE also provides graphical interfaces for each agent. These interfaces provide various data about the agents in real time, the agents they communicate with and the messages that they exchange, their behaviours, etc... An example of this default interface is presented in figure 2.3, where all the behaviours and contacts of the agent is displayed.

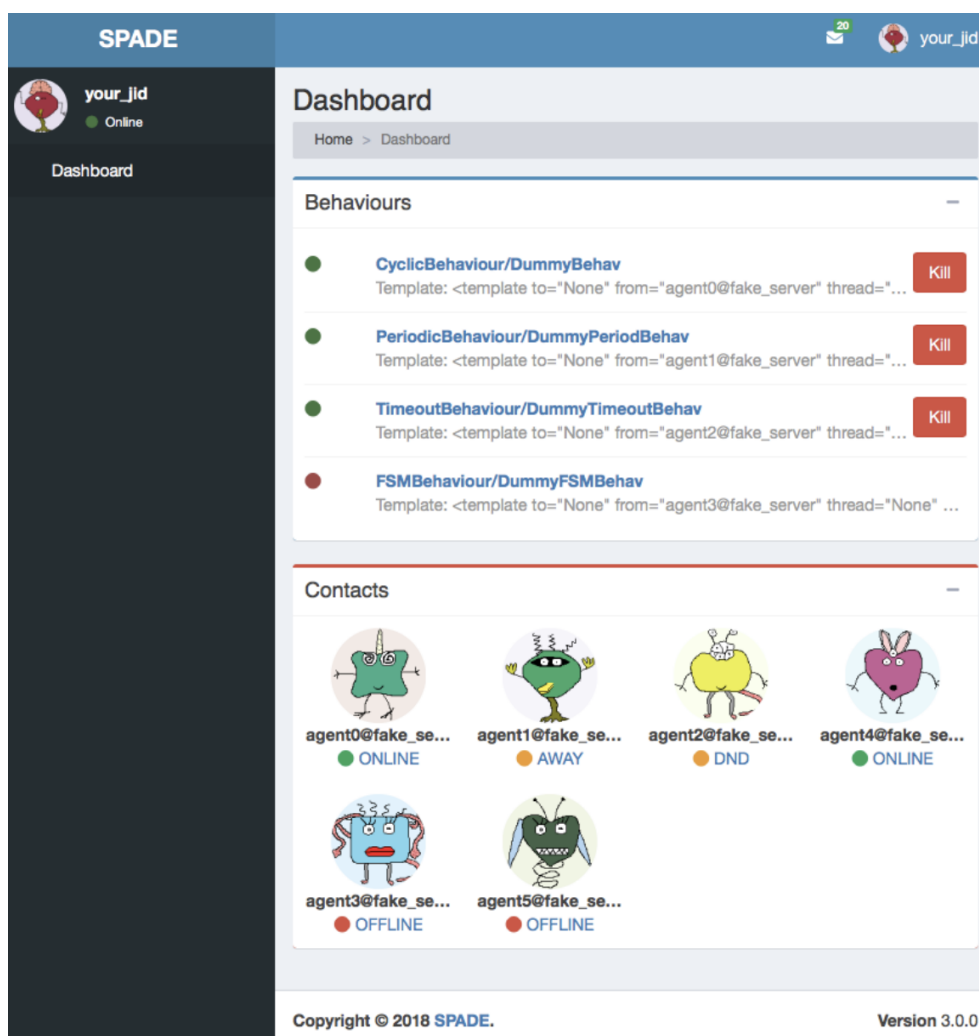


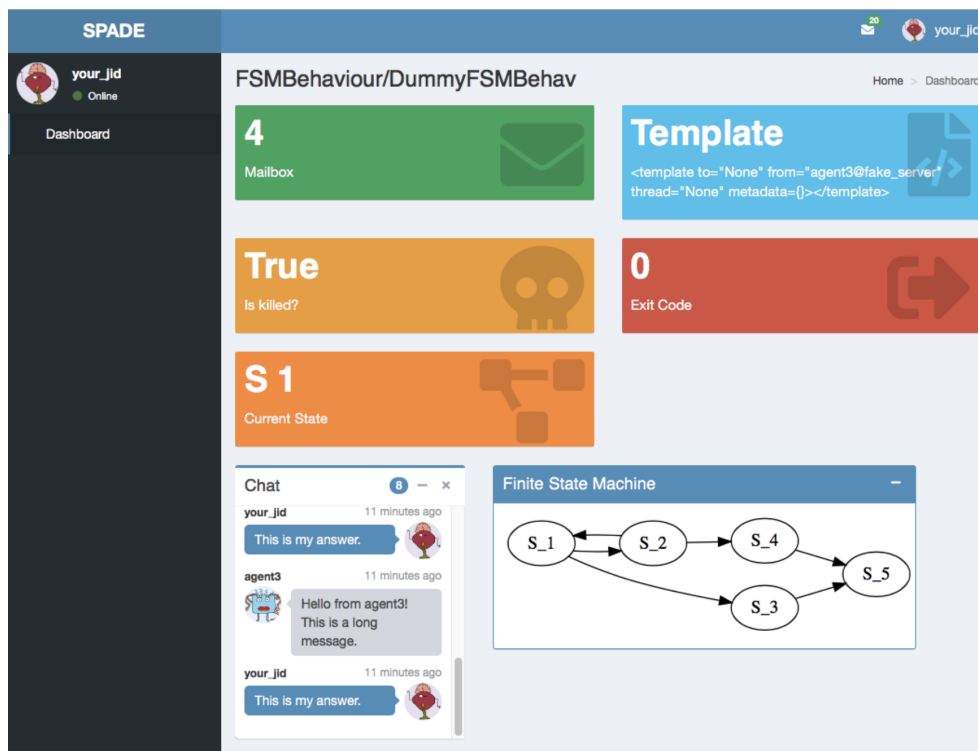
Figure 2.3: Example of a SPADE agent's default graphical interface

Details about the several behaviours of the agent can also be shown, such as in the example in figure 2.4.

In addition, SPADE allows the developer to create its custom agent graphical interfaces. Indeed, SPADE provides a scheme that allows the developer to create its own HTML file, using the Jinja<sup>6</sup> syntax. The agent must then contain a controller function,

<sup>6</sup><https://jinja.palletsprojects.com/en/3.1.x/>





**Figure 2.4:** Example of a SPADE agent behaviour's default graphical interface

that allows to send data relative to the agent constantly to the graphical interface. Note that these graphical interfaces can be used for information displaying as we discussed until now, but also for information retrieval. Indeed, the graphical interface could be a form to fill up by the user, and the data entered would then be sent to the agent that could make an adequate use of it.

## 2.2 Federated Learning

Now that the concept of Multi-Agent Systems has been introduced, it is important to understand how to relate it with the concept of Federated Learning, and how the agents in the MAS setting become clients in the FL setting.

### 2.2.1. Background on Artificial Intelligence and Machine Learning

Before diving into the details of Federated Learning, it is necessary to give a brief introduction about Artificial Intelligence and Machine Learning in general. Although it may seem like a new concept, the first program commonly seen as implementing Artificial Intelligence was released in 1956 [9].

Artificial Intelligence refers to the development of computer systems able to realize tasks which normally would require human intelligence [29]. In the case of this work, a subset of AI will be made use of : Machine Learning (ML), which "gives computers the ability to learn without explicitly being programmed" [61]. ML makes use of models, which are programs that learn from the input data and produce outputs. It is important to understand that not all of AI can be seen as ML. Indeed, for example, an algorithm

programmed to play Tic Tac Toe where the developer of the algorithm indicated the best move for each singular position is considered as AI since the algorithm plays the game perfectly without requiring the presence of a human. However, it is not ML since the algorithm didn't learn anything, it simply applied rules that were explicitly given to it. On the other hand if someone was to create an algorithm that learns from previous Tic Tac Toe games in order to be able to play, this would be ML. In this study we will focus on ML since the algorithms used will all form part of this family.

### Supervised and unsupervised learning

ML can be split into 2 categories : supervised and unsupervised learning. In the supervised setting, the ML model is trained with labeled data. In other words, the data that the ML model will train on will already contain the real value of the prediction that the model will try to output [62]. For instance if a model is trained to recognize cats from dogs on images in a supervised setting, the user will know the *answer* (whether each picture is a dog or a cat) for each piece of data that will be used to train the model.

In the unsupervised setting, the data used to train the model is unlabeled, which means the user does not know the *answer* that the model is trying to predict. The models therefore try to recognize patterns in the data, and can perform tasks such as clustering [11].

In the case of this study, we will only consider the supervised learning setting.

### Classification and regression

Two subsets of supervised ML are classification and regression. Classification refers to a problem where the goal of the model is to output a discrete variable, or *class*. For instance, the cat/dog recognition problem mentioned previously is a classification problem since there are only 2 possible outcomes.

On the other hand, regression concerns the problems where a continuous variable needs to be outputted by the model (for example the price of a product).

In this study we will focus on classification problems.

### Train-test splitting

In order to train a model, the input data is split into 2 sets : the training set and the test set. The training set is used to train the model, and the test set is then used to evaluate the quality of the model. The sizes of both sets depends on the data and the model being used for training. The size of the sets was proven to quite heavily influence the performance of the models in some cases [47].

The reason why the data is split into training and test sets is because if we would evaluate the model with the same data that we trained it on, we don't have a proper idea

of how the model reacts to data it has never seen before. If the model is good at predicting data from the training set but bad at predicting data from the test set, we say that the model is *overfitting*. In this case, the model has learned too much about the training set, to the point where it can not cope with new data [74]. We could imagine this as if the model had "*memorized by heart*" the training set instead of learning the real relation between the input data and the output.

On the other hand, when the model does not manage to capture the relationship between the input data and the output on the training set nor on the test set, this situation is called *underfitting*. In this case, it means that the model is not complex enough to capture this relationship, and needs to be adapted.

### Performance metrics

In order to evaluate a model and to be able to compare models together, performance metrics need to be defined [42]. The most simple and easily understandable performance metric for a classification problem is the accuracy, which is defined as follows :

$$Accuracy = \frac{\# \text{ of Correct Predictions}}{\# \text{ of Total Predictions}} \quad (2.1)$$

This metric gives us a number between 0 and 1 which indicates the proportion of correct predictions made in comparison to the total number of predictions.

The accuracy is first of all measured on the training set, which we will call the *training accuracy*, and then on the test set which we will call *test accuracy*. In general, the training accuracy is always higher than the test accuracy, since the models always slightly overfit the training set. The goal is that the difference between the training accuracy and the test accuracy stays quite small, in order to have a robust model.

Although there exists many other performance metrics, accuracy is the metric that will be used throughout this study. More specifically, when comparing models between themselves, test accuracy will be used.

### Model training process

The training of the model is the process during which the model learns from the input data, and learns how to make accurate predictions. As explained previously, models output predictions. In the case of a classification problem with more than 2 possible classes (called non-binary classification problems), the model outputs predictions of probabilities of the sample belonging to each class. The class with the highest probability is then considered as the output prediction.

A large amount of models contain a certain number of *weights* and *biases*, that allow to produce mathematical outputs. Weights and biases are simply numbers that describe a given model. The output of the model is a mathematical combination of all the weights and biases of the model. These weights will then be adjusted throughout the training in order to produce the best possible outputs. In order to evaluate the quality of an output during the training phase, a *Loss Function* (or Cost Function) has to be defined. This is

a function that compares the predictions of the model to the actual class of the samples. The objective is always to minimize this loss function in order to have predictions that are as close as possible to the real classes. Some of the most used Loss Functions are the MSE (Mean Squared Error) [27] or Cross-Entropy Loss [75]. Another very popular loss function is the Negative Log Likelihood (NLL). It corresponds to the negative sum of all logarithms of the outputs of the model, and the result shows how good or bad the model is at predicting the correct class in a classification case.

Once a loss function is defined, the training process works as follows :

- One sample is given as input to the model, and the output is computed
- The cost function is applied to this output, in order to see how good or bad the prediction is
- An algorithm is applied to change the weights in a way that the cost function is minimized

These steps are then applied for all samples, which then constitutes an *epoch* of training. In most cases, multiple epochs of training are required before obtaining a performant and robust model.

This algorithm applied to change the weights of the model is the key of the training process, as it is the element that actually makes the model learn from the data. Most algorithms are gradient based, which means that they compute the gradient of the cost function in regard to the weights. Indeed, the sign of the gradient will give an indication on how to modify the weights in order to minimize the loss function. The most popular algorithms used are the Stochastic Gradient Descent (SGD) [58] and Adam, which is an alternative algorithm that is not gradient based [34].

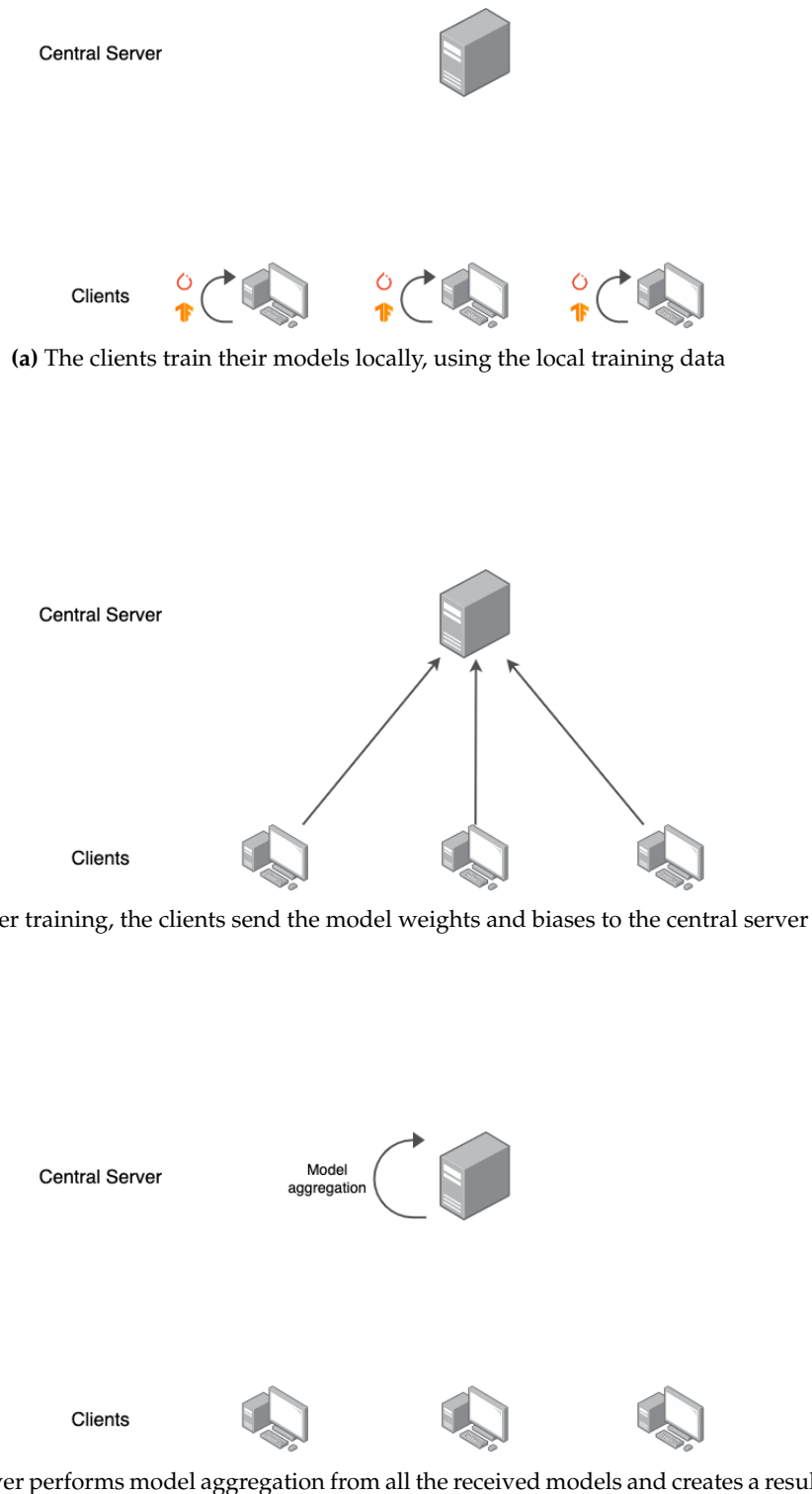
In some cases, samples are not considered one by one during the training process, but are instead grouped by batches. Each batch is processed completely before applying the cost function and changing the weights. The batch size corresponds to the number of images in a single batch.

### 2.2.2. Basic principles of Federated Learning

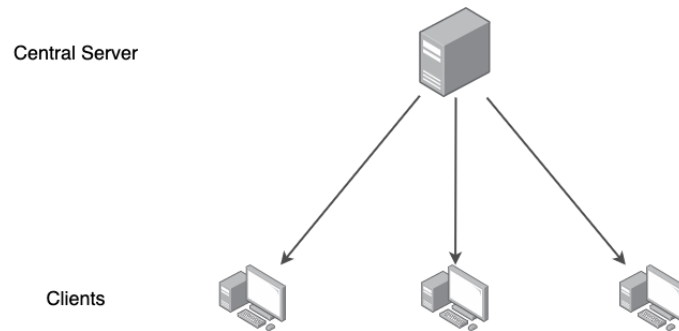
The term *Federated Learning* (FL), was first introduced by McMahan et al. [45], and defined as follows : "We term our approach Federated Learning, since the learning task is solved by a loose federation of participating devices (which we refer to as clients) which are coordinated by a central server." The main specificity of FL compared to other Artificial Intelligence techniques is that it uses multiple devices, and model training is not made on one sole device. This implies that devices have to be able to communicate with the central server, which means some kind of communication network needs to be in place for FL to be used.

FL exploits the resources of each user device in order to train a Machine Learning model, by keeping the training data decentralized. Indeed, each user device performs a local training using the data at his disposal, and then sends information (model weights and biases) to the central server. The central server then performs a step called *Model*

*Aggregation*, which consists in combining the different models that were received, and to have one resulting model as a result. There are multiple model aggregation techniques, some of which will be presented throughout this paper. This resulting model is then used to measure the accuracy on the test set. The central server then sends the updated weights obtained after model aggregation to the clients, and the next round of local training starts based on these new weights. The different steps of a Federated Learning training iteration are presented in Figure 2.5.



(c) The central server performs model aggregation from all the received models and creates a resulting model



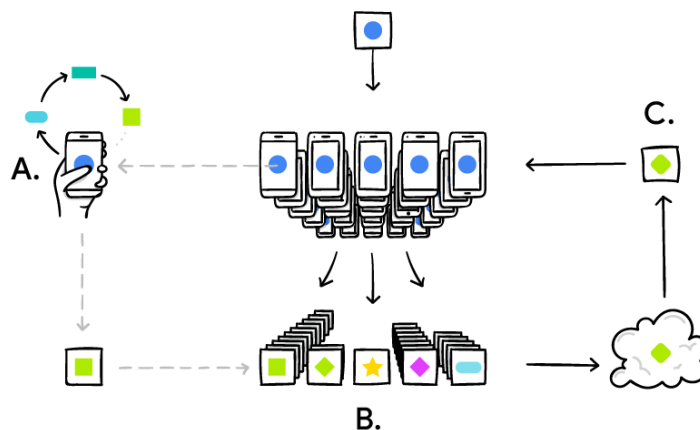
(d) The central server sends the model resulting from model aggregation to the clients

**Figure 2.5:** Steps of a Federated Learning training round

FL presents a lot of advantages compared to traditional AI techniques. First of all, FL embodies the principle of data minimization as each user trains its model on a small portion of the total data used throughout the process. The fact of not overpowering ML models with large quantities of data and keeping the amount data minimal in model training has proven to be way more efficient [20]. Furthermore, FL can present an advantage in terms of execution times. Indeed, the model training phase is much faster in this scenario as it is made with a smaller quantity of data, and the model aggregation is usually a quickly executed phase. The limitations of FL and of the gain in computational time will be further studied in chapter 5.

FL allows to solve a tackle a serious challenge : analyze and learn from data distributed among multiple devices or users by running Machine Learning models while maintaining the privacy and not exposing this data. Indeed, exploiting data in big data centers that collect information from all clients, even anonymously, has proven to be at high risk of breaching data privacy [66]. By using FL, the raw training data never leaves the user's device, as the training is performed locally. The communications between the user devices and the central server only contain model weights and biases, which do not reflect directly the training data and preserves the user's privacy.

This is for example a problem tackled by Google with Gboard, a keyboard developed by Google capable of suggesting the next work depending on the context. In order to guarantee privacy for all users and still be able to learn from all user experiences and history of typing, FL was used in order to train locally the models on each user device, and then simply sharing model parameters and weights to a central server [46]. The different steps illustrating how FL allows to learn from user input in the case of Gboard is described in Figure 2.6. In the step A, the user input is collected and a model is trained locally. Next, in step B, all the user devices send the learned data to a central server. This server makes a model aggregation in step C, and updates the models in each individual user device.



**Figure 2.6:** Usage of Federated Learning in Gboard. This picture was taken from [46]

### 2.2.3. Centralized and Decentralized Federated Learning

The Federated Learning principles explained in section 2.2.2 refer to what would be more specifically called *Centralized Federated Learning*. This term of course refers to the presence of a central server, responsible for performing the model aggregation once all the clients perform a training step. However, this central server presents a single point of failure (SPoF) and could also be a bottleneck because of all the messages arriving from the multiple clients at the same time. Indeed, it was proven that with a large number of clients, the centralized architecture could face serious bottlenecks due to communication overflows and lead to a severe lack of performance [39]. Therefore, some alternatives to Centralized Federated Learning have been advanced, in order to mitigate the disadvantages of the central server [68].

The key idea of *Decentralized Federated Learning* is that the central server is removed, and that the clients directly communicate one with another using a peer-to-peer architecture. This therefore means that the clients are themselves performing the model aggregation step. The communication topology is therefore a bit more complex than in the Centralized Federated Learning scenario, as it is represented by a graph where the nodes are the different clients and the edges connect the clients that can communicate one to another. Not all the nodes have to be connected to each other, the graph is made completely arbitrarily and depends on the context of the execution. Therefore, all clients do not necessarily have the same number of neighbours in the graph, and therefore do not all connect with the same amount of clients. This creates a sort of asymmetry that does not exist in the context of Centralized Federated Learning. Note that the graphs used in the decentralized setting are usually undirected (if a client  $b$  is the neighbour of the client  $a$ , then client  $a$  is also the neighbour of client  $b$ ), but cases with directed graphs have also been studied. For instance, this can be useful when studying single-sided trust social networks (if a client  $a$  trusts client  $b$ , client  $b$  may not trust client  $a$ ) [24].

In this decentralized setting, clients can be in different states :

- Training State : the client is locally training his model
- Sending State : the client is sending his model's weights and biases to his neighbours

- Receiving State : the client is waiting for his neighbours to send him their model's weights and biases

The Decentralized Federated Learning setting can further be divided into two different versions : Synchronous Decentralized Federated Learning and Asynchronous Decentralized Federated Learning [41].

### Synchronous Decentralized Federated Learning

In the Synchronous Decentralized Federated Learning (SDFL) setting, a round of training can be described as follows :

- All clients start training their local models, using their local training data
- As soon as a client is done training, he sends his model weights and biases to all his neighbours in the graph
- Each client waits until they have received the weights and biases of all their neighbours, and then perform a local model aggregation
- Once the model aggregation has been performed, the local training starts again

In order to illustrate the Synchronous Decentralized Federated Learning more clearly, an example is given using 4 clients, connected to each other as presented in figure 2.7. The several steps of an iteration in the SDFL setting in the case of these 4 agents is presented in figure 2.8.

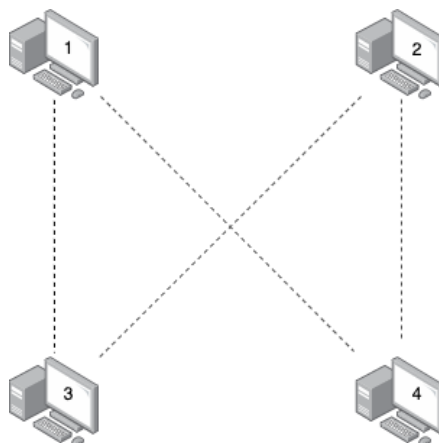
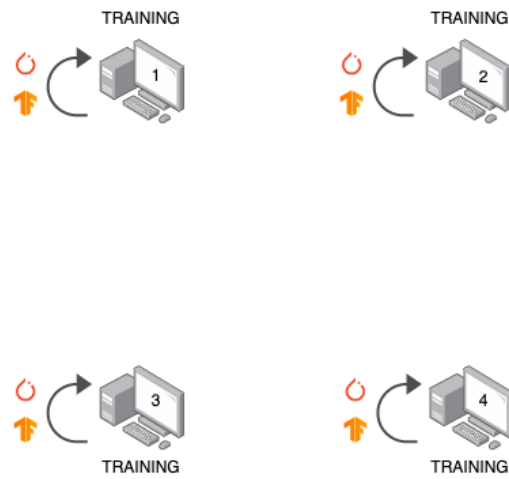
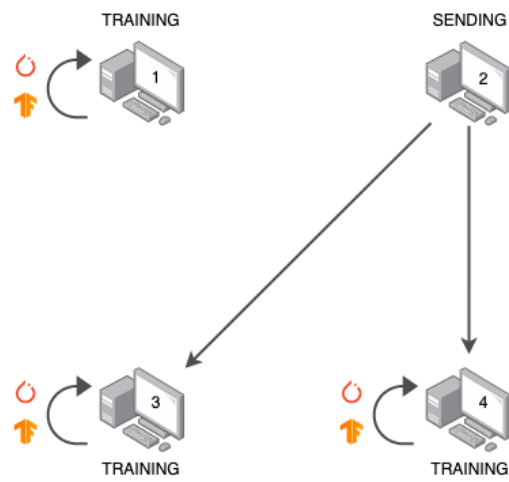


Figure 2.7: Graph representing the agents and their connections

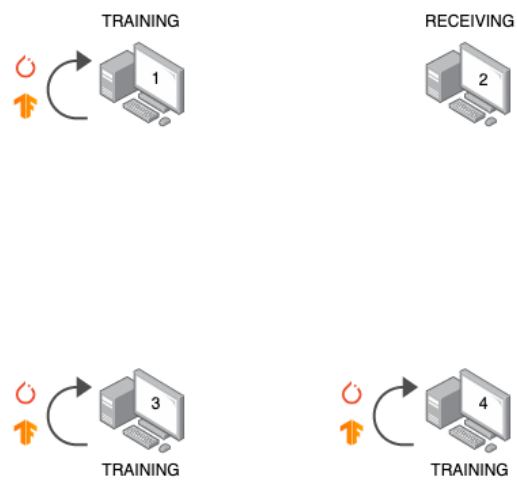




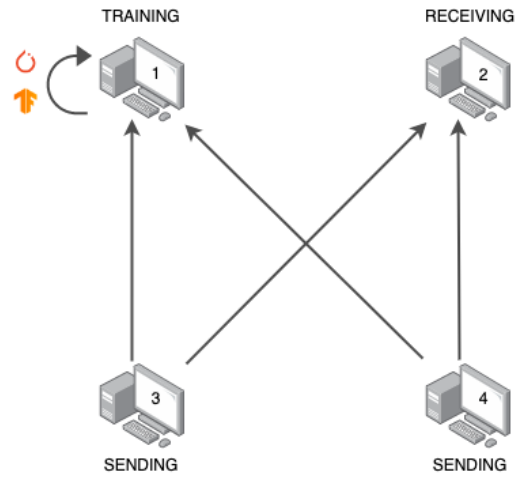
(a) The clients train their models locally, using the local training data



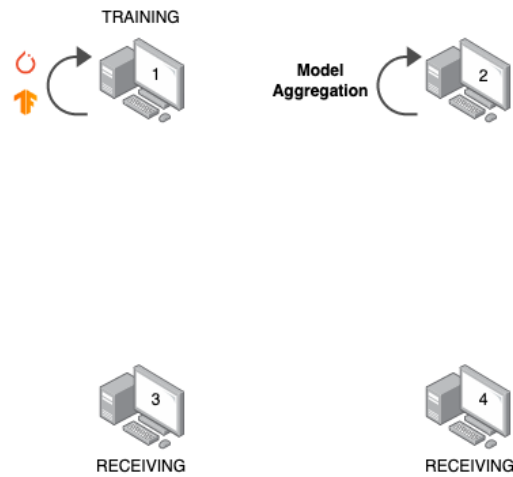
(b) Client 2 is done training, he sends his model weights and biases to his neighbours



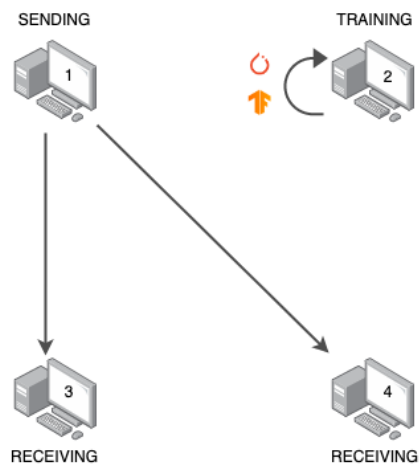
(c) Client 2 now waits for the weights and biases of his neighbours (clients 3 and 4)



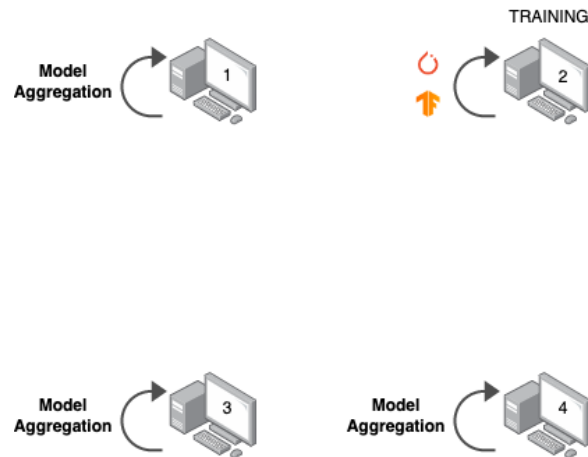
(d) Clients 3 and 4 are done training and send their model weights and biases to their neighbours



(e) Client 2 now received the weights from all his neighbours and performs model aggregation



(f) Client 1 is done with training and sends his weights and biases, client 2 starts training again



(g) Clients 1, 3 and 4 have now received all the weights and biases from their neighbours and perform model aggregation

**Figure 2.8:** Steps of a Synchronous Decentralized Federated Learning training round

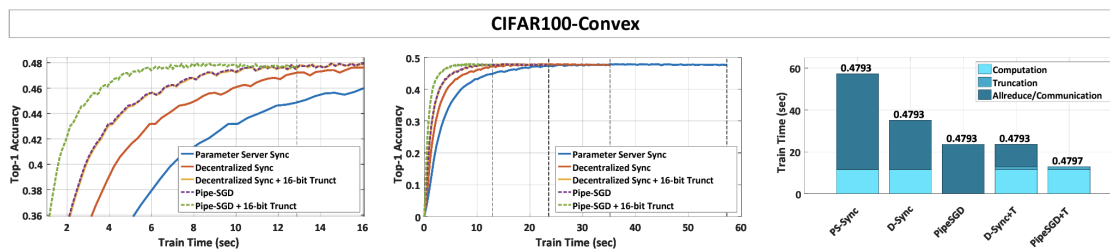
As mentioned previously, the SDFL setting offers multiple advantages compared to the Centralized FL setting, mainly brought by the removal of the central server that represents a Single Point of Failure as well as a bottleneck with the important traffic of incoming messages.

However, the SDFL architecture presents one major inconvenient, which is the large amounts of time that the agents spend waiting for their neighbour's weights. Indeed, referring to the example in Figure 2.8, the client 2 finishes training first, and has to wait a long time until the clients 3 and 4 also finish their training round and send their weights. After a training round, an agent has to wait for the model weights of all of its neighbours and can not start another training round in the meantime due to the synchronous character of the architecture. A lot of execution time is therefore lost as the agents spend a non negligible time waiting, and this alters the overall efficiency of the SDFL solution.

The first studies of Synchronous Decentralized Federated Learning were made by Lan et al. [35] and by Sirb and Ye [64]. These studies give a theoretical proof that Synchronous Decentralized Federated Learning leads to convergence asymptotically, but no theoretical speedup could be proven compared to the centralized architecture. Indeed, the speedup depends on a number of factors, some of which are more practical than theoretical (such as communication delays, training times,...).

Li et al. [38] then proposed a new solution, actually implementing Synchronous Decentralized Federated Learning and applying various techniques in order to compress the data that is sent between the devices. In the end, experiments showed that the decentralized architecture did present a clear advantage over the centralized architecture. For example, figure 2.9 shows the results of one of the experiments carried out by Li et al., where we can clearly see that the implementation represented by the green curve (*Pipe-SGD + 16-bit Trunc*) is the more efficient. Indeed, it reaches higher accuracies than other models in the same amount of time, before stagnating at 0.48. This implementation is in reality a variant of Synchronous Decentralized Federated Learning, where pipelining was used in order to be able to train the model and communicate at the same time. The blue line on the graphs corresponds to the centralized architecture, and we can see that

all the decentralized solutions clearly outperform the centralized one.



**Figure 2.9:** Comparison between a centralized architecture and various implementations of asynchronous architectures. This figure was taken from [38].

In 2018, He et al. [25] create COLA (**C**ommunication-**E**fficient **D**ecentralized **L**inear **L**earning), which has been proven to converge and presented significantly better experimental results than the centralized setting. Nedic et al. [49] adapted this solution to time-changing networks, which means that the network is dynamic, nodes can enter and leave the network whenever. The convergence of the solution was still proven asymptotically, and proved to have better results compared to the centralized architecture.

Let us not forget that a centralized architecture also provides some advantages. In particular, the devices used by clients are sometimes of a different nature, and establishing a communication scheme between themselves can prove to be complicated, whereas if all the agents communicate only with the central agent the communication protocol is easier to establish. Furthermore, the central server allows to change the model aggregation characteristics more easily. If a different type of consensus was to be applied, it should only be changed in the central server as it is never performed by the agents themselves. Also, the central server can in some cases be helpful in terms of security and privacy. Indeed, all the agents communicate only to the server, therefore no information relative to one agent is directly sent to another agent. The security and privacy concerns in Federated Learning is studied by Kairouz et al. in [31].

In general, it is now commonly accepted that Synchronous Decentralized Federated Learning is implementable and does present advantages in most use cases compared to centralized architectures.

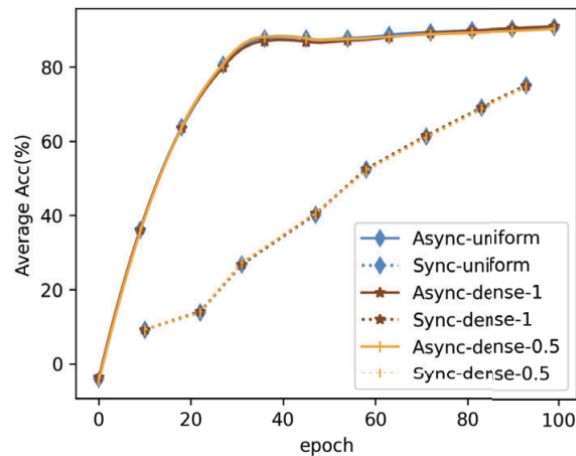
### Asynchronous Decentralized Federated Learning

As mentioned previously, the Asynchronous Decentralized Federated Learning algorithm is a new algorithm that was developed throughout this study. However, some asynchronous Federated Learning algorithms have been studied in the literature, but they do not operate the same way as the algorithm that will be implemented in this study. The Asynchronous Decentralized Federated Learning implementation is an area that is in general less studied, because it is more complex mathematically speaking, and also harder to implement. There are not a lot of implementations that have been created, to our knowledge, of this algorithm.

In 2022, Chen et al. implement an Asynchronous Decentralized Federated Learning algorithm [15] based on the FODAC (First-Order Dynamic Average Consensus) [76] con-

sensus algorithm. The clients use the models of all their neighbours during the model aggregation phase, but in an asynchronous manner. There is still, implicitly, some waiting times involved in this architecture, although the model aggregation algorithm is adapted to an asynchronous setting.

The results obtained in this study were quite convincing, as shown in figure 2.10.



**Figure 2.10:** Results of experiments on the MNIST dataset using the Asynchronous Decentralized Federated Learning algorithm described in [15]. This figure was taken from [15].

We can clearly notice that the asynchronous algorithms perform way better than the synchronous ones in this experiment. However, these results were obtained through simulations, which means that although the results are correct and analyzable, no communication-related issues or hardware issues are taken into consideration.

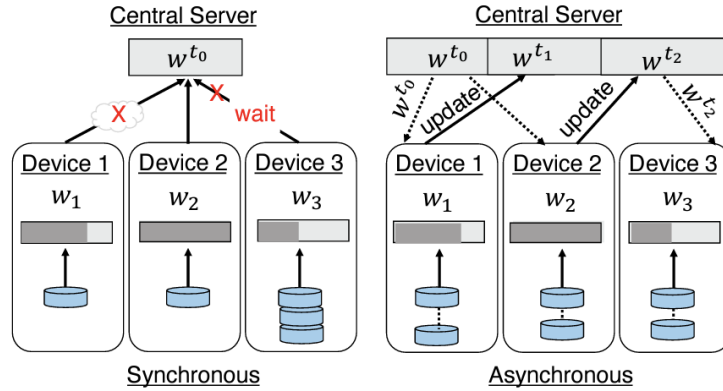
Also, Liu et al. have designed a system in order to detect faults on Power Voltage stations [41]. The results in simulation show very good results, and the asynchronous aspect of the algorithm fits perfectly to the Power Voltage station situation. However, just like in the study carried out by Chen et al. [15], the consensus algorithm during the model aggregation phase is made only when the weights of all the neighbour agents are available.

However, creating a solution that implements Asynchronous Decentralized Federated Learning is still, to our knowledge, non-existent. Furthermore, an algorithm with a consensus algorithm that updates the model after every message received by a neighbouring agent, instead of using all neighbours' models at each model aggregation as done by Chen et al. [15] does not seem to be existing either. Therefore, the goal of this study is to implement this solution with SPADE agents with a fully decentralized consensus algorithm, and to evaluate this solution compared to other FL techniques.

#### 2.2.4. Asynchronous Centralized Federated Learning

Some work has been made in the field of Asynchronous Centralized Federated Learning for instance, where there is still the presence of a central server but where the clients communicate with this model in an asynchronous way and the model aggregation is also performed in an asynchronous fashion [14]. This is an interesting subject for this

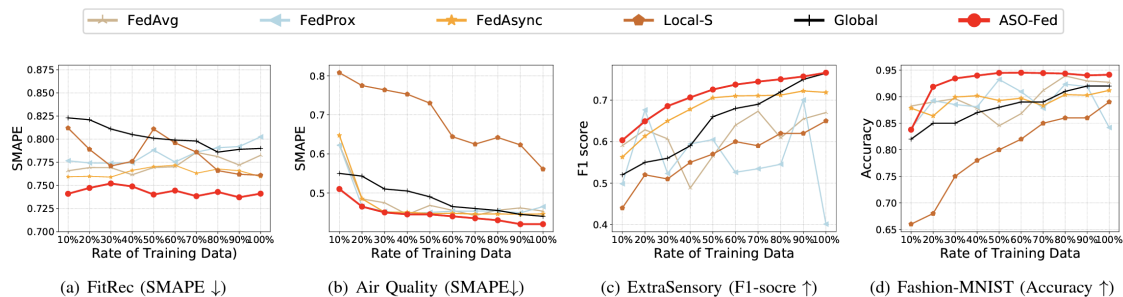
work although we are focusing on decentralized settings. Indeed, the way to handle the asynchronous aspect in a centralized setting could be useful to implement Asynchronous Decentralized Federated Learning. The difference between the synchronous and asynchronous versions of Centralized Federated Learning are shown in figure 2.11.



**Figure 2.11:** Comparison between a synchronous and asynchronous implementation of Centralized Federated Learning. This figure was taken from [14].

We can see that the asynchronous implementation improves the solution as if one device is down, or takes too long, the other devices can still communicate with the central server, which performs model aggregation and returns updated weights to all agents. Chen et al. developed an Asynchronous Centralized Federated Learning algorithm called ASO-Fed. They made several experiments (the results are presented in figure 2.12) in order to evaluate this solution, using as metrics the following metrics :

- SMAPE (Symmetric Mean Absolute Percentage Error) : a higher value means a higher error and therefore a lower accuracy [43]
- F1-Score : compares the precision (number of true positive results divided by the number of all positive results) and the recall (the number of true positive results divided by the number of all samples that should have been identified as positive) [40]
- Accuracy : defined in equation 2.1



**Figure 2.12:** Results of experiments on the following datasets : FirRec, Air Quality, ExtraSensory, Fashion-MNIST, with several Federated Learning algorithms. This figure was taken from [14].

In all 4 experiments, we notice that the ASO-Fed algorithm is the most efficient and performant. This shows that Asynchronous Centralized Federated Learning can present

some great advantages over the synchronous methods, but the central server still remains a Single Point of Failure (SPoF), which lead to the studies of asynchronous solutions.

## 2.3 Consensus-Based Learning

Now that the concepts of Multi-Agent Systems and Federated Learning have been explained, and that the advances made in the research of decentralized and asynchronous implementations have been explored, it is important to focus on the Model Aggregation step, which is the crucial part of Federated Learning. This phase is extremely important as it merges several ML models together with the goal to produce an output model. This step makes use of the consensus principle, which allows to obtain one output model that merges the information contained in all input models. Federated Learning can therefore be seen as a form of Consensus-Based Learning, this chapter intends to define formally how the consensus is performed during the model aggregation phase, and to put in evidence why this technique allows to reach a consensus in the whole network of agents.

### 2.3.1. Theory of Consensus

The term *consensus* in the context of a network of agents is defined in [51] as the fact to "reach an agreement regarding a certain quantity of interest that depends on the state of all agents". In the case of this work, the quantity of interest represents all the weights constituting the models of the agents.

In order to give a formal definition of the consensus in the context of a network of agents, we will define the directed network graph of  $n$  agents as  $G = (V, E)$  where  $V = 1, 2, \dots, n$  is the set of all vertices of the graph and  $E \subseteq V \times V$  is the set of all edges of the graph (the same notation as in [51] is used). We will call  $A = \{a_{ij}\}$  the adjacency matrix of  $G$ , which is defined as follows :

$$a_{ij} = \begin{cases} 1, & \text{if } (i, j) \in E \\ 0, & \text{otherwise} \end{cases} \quad (2.2)$$

We also define  $N_i$  as the set of all neighbours of agent  $i$ , such as  $N_i = \{j \in V : (i, j) \in E\}$ . The degree of a node  $i$  corresponds to the number of neighbours that agent  $i$  has, which can be denoted by  $|N_i|$  or  $deg(i)$ . We can therefore define the degree matrix of the graph  $G$  as  $D = \{D_{ij}\}$ , which is defined such as :

$$D_{ij} = \begin{cases} deg(i), & \text{if } i = j \\ 0, & \text{otherwise} \end{cases} \quad (2.3)$$

Furthermore, we define the Laplacian of graph  $G$  as :

$$L = D - A \quad (2.4)$$

Since a consensus is performed on a quantity of interest, we will define the value of an agent  $i$  as  $x_i$  for all  $i \in V$ . In more general terms, we will consider  $x$  as the ensemble of all values  $x_i$  of all nodes  $i$ ,  $x = (x_1, x_2, \dots, x_n)^T$ . We say that consensus was reached when there exists a value  $x^*$  such as  $x_i = x^* \forall i \in V$  [59].

### Continuous-Time Consensus

In a continuous-time model, the consensus algorithm can be expressed by the following equation [51] :

$$\dot{x}_i(t) = \sum_{j \in N_i} (x_j(t) - x_i(t)) \quad (2.5)$$

where  $\dot{x}_i(t)$  represents the derivative of  $x_i(t)$  in regard to  $t$ .

This equation can also be rewritten in a matrix form, as follows :

$$\dot{x} = -Lx \quad (2.6)$$

where  $L$  is the Laplacian of graph  $G$ , as shown in equation 2.4. In this case,  $\dot{x} = (\dot{x}_1, \dot{x}_2, \dots, \dot{x}_n)^T$  and  $x = (x_1, x_2, \dots, x_n)^T$ .

It can be proven that if all  $n$  nodes of a connected graph  $G$  follow this algorithm, then all nodes asymptotically reach a consensus value  $x^*$  such as  $x_i = x^* \forall i \in V$  [59]. Furthermore, this final value is actually the average of all initial values of all nodes of the graph :

$$x^* = \lim_{t \rightarrow -\infty} x(t) = \frac{\sum_{i \in V} x_i(0)}{n} \quad (2.7)$$

### Discrete-Time Consensus

In the case of this work, we will focus on the iterative version of the consensus algorithm, as presented in [51]. In this case, we will define  $k \geq 0$  as the number of iterations of the consensus algorithm that have been performed. In general terms, the iterative form of the consensus algorithm can be formalized as follows :

$$x_i(k+1) = x_i(k) + u_i(k) \quad \forall i \in V \quad (2.8)$$

This equation illustrates that the changes made to a certain value  $x_i$  after  $k+1$  iterations depends on the value of  $x_i$  after  $k$  iterations which is altered by another term  $u_i$ . By expanding the second term of the equation, we obtain the more detailed expression of the iterative consensus algorithm :

$$x_i(k+1) = x_i(k) + \epsilon \sum_{j \in N_i} (x_j(k) - x_i(k)) \quad \forall i \in V \quad (2.9)$$

where  $\epsilon > 0$  represents the *step-size*. We therefore notice that the value of  $x_i(k+1)$  is influenced by the values of all the neighbours of node  $i$ . Indeed, since the sum is made on the set  $N_i$  of all neighbours of node  $i$ , the values of each of the neighbours are taken into account. Furthermore, we notice that the step-size  $\epsilon$  gives an indication on how much the neighbours' values influence the new value  $x_i(k+1)$  of node  $i$ . Indeed, if the value of  $\epsilon$  is very low, the  $u_i$  term has a limited influence on the computation of  $x_i(k+1)$ , which therefore relies heavily on the previous iteration value  $x_i(k)$ . On the other hand, if the *epsilon* value is high,  $x_i(k+1)$  is more influenced by its neighbour's values.

Equation 2.9 can be rewritten in the following way using a matrix notation :



$$x(k+1) = (I - \epsilon L)x(k) \quad (2.10)$$

where  $L$  is the Laplacian of the graph  $G$ . The term  $(I - \epsilon L)$  can be referred to as the Perron matrix, and denoted by  $P$ . Equation 2.10 can therefore be rewritten as :

$$x(k+1) = Px(k) \quad (2.11)$$

We define  $\Delta = \max(D_{ij})$  as the maximum degree of graph  $G$ , which corresponds to the maximum number of neighbours that any node in  $G$  has. It can be proven that if a network of  $n$  agents that is represented by a strongly connected digraph  $G$  follows the consensus equation presented in 2.9 and that  $0 < \epsilon < 1/\Delta$ , then [51]:

- A consensus is asymptotically reached for all agents' initial states;
- The final value of each agent is a linear combination of all initial values of all agents such as  $x^* = \sum_i w_i x_i(0)$ . The  $w_i$  values are real coefficients such as  $\sum_i w_i = 1$ ;
- If  $G$  is a balanced digraph, then the final value of each agent is the average of all agents' initial values such as  $x^* = (\sum_i x_i(0))/n$ .

In order to understand this theorem, it is important to define what a *strongly connected digraph* is, which is a necessary condition for  $G$  for the theorem to be applicable. A digraph is a directed graph, which means that all edges have a certain direction, they go from one node to another. A digraph is strongly connected if and only if it is possible to reach all nodes starting from any node and traversing the edges of the graph in their appropriate direction. Therefore, if  $G$  respects this condition, the previously stated theorem guarantees that a consensus will be reached asymptotically.

Furthermore, it is important to clarify the 3rd point of the theorem, in order to understand when the obtained consensus is an average-consensus. A digraph is balanced, if and only if for every node  $i \in V$ , the amount of edges entering the node and the amount of edges leaving the node are equal. In more formal terms, a balanced digraph can be defined as follows :

$$\sum_i a_{ij} = \sum_i a_{ji} \quad \forall i \in V \quad (2.12)$$

In the case of this study, we will systematically work with undirected graphs, which are graphs where all edges are bidirectional. Indeed, when an agent can connect to his neighbour, he must be able to answer to him, so they are both consequently connected to each other. This, however, does not go against the definition of a digraph, since an undirected graph is a particular case of a digraph where for every edge  $(i, j) \in E$ , there exists another edge  $(j, i) \in E$ . Furthermore, an undirected graph is always balanced. Indeed, since all edges are bidirectional, it means that the number of entering edges is by definition always equal to the number of exiting edges for each node. Therefore, the only conditions to respect throughout this study in order to obtain an average-consensus (a consensus where the final value is the average of all agents' initial values) is to have a strongly connected network graph and to satisfy the inequality  $0 < \epsilon < 1/\Delta$ .

---

## 2.4 Proposition

---

From this review of the state of the art, we realize that there is no existing Asynchronous Decentralized Federated Learning algorithm where the agents are never waiting. We could therefore imagine an algorithm where the model aggregation is made by an agent, only considering one of its neighbour's model at a time. Indeed, this would mean that every time an agent receives the model weights of a neighbour, he applies the consensus to perform model aggregation with just these received weights. This eliminates totally the waiting times as the agents never have to wait to be in possession of all the neighbour's weights.

As we have seen, there seems still not to exist a proper implementation of Asynchronous Decentralized Federated Learning. Some studies have been conducted through simulations, but no real implementations have been compared to the synchronous setting. Therefore we could create a concrete solution that implements this algorithm and compare it to synchronous versions. It would also be useful to create an application out of this implementation, that would be easy to use and that would allow any user to very easily launch, control, monitor and evaluate the performance of different agents executing the ADFL algorithm. Our hypothesis is that ADFL will, at least in some use cases, present advantages in terms of performance compared to existing solutions. Therefore, this application could be used in practical cases in order to apply Federated Learning in a more efficient way.

---

## 2.5 Starting point of the implementation

---

Before the start of this work, a large part of the desired implementation concerning Synchronous Decentralized Federated Learning was already implemented. Indeed, Pr. Carlos Carrascosa Casamayor and Pr. Jaime Andrés Rincón Arango had started implementing the code for the Synchronous Decentralized Federated Learning implementation. The implementation was not completely done, some code had to be added to create logs and conduct experiments, but the major part was already done.

Concerning the Asynchronous Decentralized Federated Learning however, the code was entirely produced during this work, although taking the Synchronous Decentralized Federated Learning code as a starting point.

However, the professors had created and tested Google Colab<sup>7</sup> notebooks that implement simulations of Synchronous Decentralized Federated Learning and these were adapted to Asynchronous Decentralized Federated Learning. These files were executed to prove the convergence of the algorithms before starting the Python implementations, and were very useful for the comprehension of the subject before starting the development.

---

<sup>7</sup><https://colab.research.google.com/>

## 2.6 Preliminary analysis

Before starting to develop the application to implement the ADFL algorithm, a preliminary analysis was conducted in order to identify the key requirements as well as the risks involved in the development.

### 2.6.1. Definition of requirements

Concerning the theoretical work to conduct in order to produce the ADFL algorithm, some of the main requirements are presented in table 2.1, where the importance of each requirement is also presented.

#	Name	Description	Importance (1-5)
1	Eliminate the waiting time	The clients of the network should never have to be waiting, without doing anything, for the weights of their neighbours before doing model aggregation.	5
2	Asymptotically reach consensus	The values of the clients should asymptotically reach the same value, the models should converge.	5
3	Adapt to dynamic networks	The clients should be able to adapt when agents enter or leave the network.	4
4	No training time waste	The time clients spent training their model should never be wasted, all the training should be fully exploited.	3
5	Message reception at anytime	The clients should be prepared to receive messages at anytime, no incoming messages can be discarded.	4
6	Realism	The existing technologies should be kept in mind when developing the algorithm, since it will have to be implemented in a real application.	5

Table 2.1: Requirements for the development of the ADFL algorithm

These requirements were always kept track of all along the project, which helped guide the work and prioritize certain tasks. As we see, the most important requirements is to develop a feasible algorithm that works as expected (so reaches consensus) and that differentiates itself from SDFL by not having any waiting times (fully asynchronous).

Concerning the development of the application that implements ADFL, the main requirements are presented in table 2.2.

#	Name	Description	Importance (1-5)
1	Implement the ADFL algorithm	The application should enable a network of agents to run the developed ADFL algorithm correctly.	5
2	Adapt to dynamic networks	The agents should be able to adapt to new agents entering the network or to agents leaving the network.	4
3	Agent Graphical Interface	Present a clear Graphical Interface for each agent that displays the performance of the solution	4
4	Launcher agent	Develop a launcher agent that allows the user to enter all the specifications of the execution before executing the agents of the network.	3
5	Launcher Agent Graphical Interface	Present a clear Graphical Interface for the launcher agent where the user can enter the specifications.	3
6	Enable measures and analysis	The solution should allow to easily make studies and conclusions about the performance of the solution.	4
7	Easy to launch	The application should be easy to launch by any user, installing requirements should not be complicated.	3
8	Easy to use	A clear description of how to use the application should be given to the user.	3

**Table 2.2:** Requirements for the development of the application implementing the ADFL algorithm

### 2.6.2. Risk analysis

There are also a few risks involved in the implementation of the application that implements ADFL, which are listed in table 2.3. For each one, a way to mitigate the risk is also presented.

#	Risk	Probability	Consequences	Mitigation
1	A user can't execute the solution because he doesn't have the correct Operating System or version of Python installed	High	High	Containerize the solution so that the execution is platform independent
2	Losing all the progress of the implementation because of a technical failure or a mistake	Low	High	Use a Version Control solution such as Git during the development
3	Not having a finished product at the deadline of the project	Low	High	Make a work plan estimating the number of hours per phase and keep track of the progress of the implementation throughout the development
4	Creating code that is not understandable by other researchers who want to modify the solution	Medium	Medium	Respect good coding principles such as PEP8 and document the code
5	Presence of bugs in the code that make the application unusable	Medium	High	Test the code thoroughly and experiment all the possible settings
6	Lack of staff during a long period, unavailable supervisors	Low	Medium	Always think a step ahead, ask information about the next steps at each meeting in order to always have something to work on
7	Third-party libraries not working as expected	Low	High	Research alternative libraries in case a library does not work properly or is outdated
8	Unavailability of the computer laboratories for making experiments	High	Medium	Prioritize the most important experiments when the laboratories are available

**Table 2.3:** Risks involved in the development of the application and measures to mitigate them



---

---

## CHAPTER 3

# Elaboration of the solution

---

### 3.1 Work Plan

---

In order to efficiently build the solution to this study and document it, a work plan was made, it is presented in table 3.1. This allows to separate the different steps of the work and associate a number of hours to each one of them.

Note that the number of working hours for each phase was just an estimation, and that it is difficult to say if the actual number of working hours were close to these predictions or not.

Also, this work plan does contain some phases that are carried out in parallel. For example, the writing of the final report was carried out throughout the whole work, and was not left until the end. Indeed, it is important to have clear conclusions for each step of the project and to document it as the project advances.

### 3.2 Creating the Asynchronous Decentralized Federated Learning algorithm

---

In order to create and specify the execution pattern for the Asynchronous Decentralized Federated Learning algorithm, the first step is to explain the general scheme of the algorithm, and the different steps that constitute one iteration of this algorithm. Next, the consensus algorithm of the model aggregation phase must be adapted to this situation.

#### 3.2.1. Asynchronous Decentralized Federated Learning scheme

The Asynchronous Decentralized Federated Learning setting differentiates itself from the synchronous version presented in section 2.2.3 by the absence of waiting times after an agent finishes a local training of its model and sends its weights to its neighbours in the agent network graph. Indeed, by the asynchronous nature of this solution, the model aggregation is made gradually everytime the model weights of a neighbour agent are

Phase	Description	Predicted working hours	Difficulty
Research and documentation	Getting to know the principles of Federated Learning, Consensus-Based Learning, Multi-Agent Systems and SPADE based on academic papers, articles and official documentation.	30	Medium
SDFL implementation	Complete the existing code in order to implement the SDFL algorithm using SPADE agents. This phase also contains a lot of documentation to track the progress of the phase and establish conclusions as the progress advances.	40	Medium
Create the ADFL algorithm	Create and specify all the components of the ADFL algorithm. This phase includes a lot of research and investigation, as well as theoretical simulations to justify the elaboration of the solution.	80	High
ADFL implementation	Create the code that implements the ADFL algorithm using SPADE agents, starting from the code written for the SDFL algorithm. This phase also contains a lot of documentation to track the progress of the phase and establish conclusions as the progress advances.	100	High
Create a Docker container of the ADFL implementation	Creating a Docker image of the ADFL solution and publishing it on Docker Hub.	4	Low
Refactoring and documentation of the ADFL code	Refactoring the code and making sure that the PEP8 principles were respected.	5	Low
Testing the implementation	Carry out various experiments testing the SDFL and ADFL algorithms (mainly ADFL), compare the obtained results and establish conclusions. Some of the experiments required using the DSIC computer laboratories.	60	Medium
Writing the academic paper	Write a part concerning the ADFL implementation with SPADE agents of the academic paper [12] sent to the AAMAS conference.	20	Medium
Writing the final report	Write the final Master Thesis Report. This phase was carried out throughout the whole work and was not left as a final task.	60	Low

**Table 3.1:** Work plan. Each phase is described and is attributed a certain number of working hours as well as a difficulty.

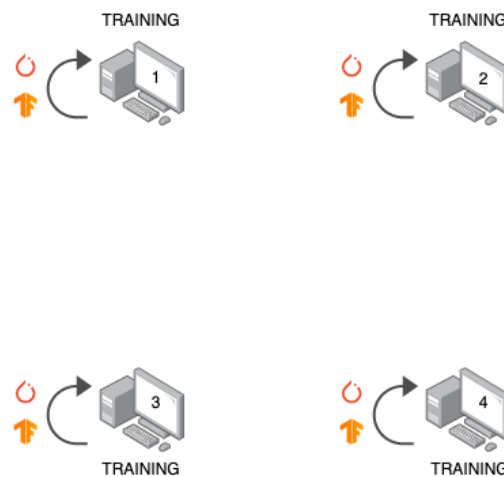


received, without interrupting the training process. This solution allows to avoid the loss of execution time present in the synchronous solution when the agents are waiting for their neighbour's responses.

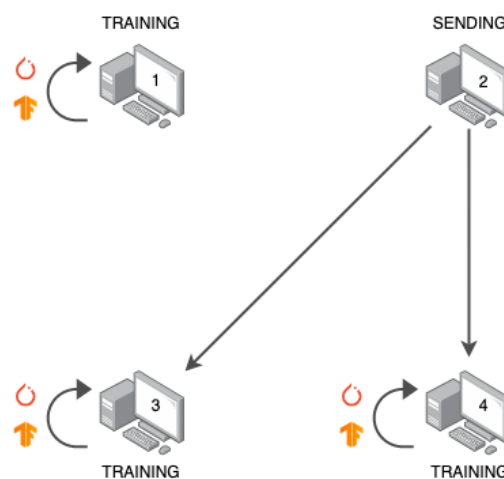
In the Asynchronous Decentralized Federated Learning setting, a round of training can therefore be described as follows :

- All clients start training their local models, using their local training data
- As soon as a client is done training, he sends his model weights and biases to one random neighbour in the graph, and this neighbour responds to him with its current weights and biases
- He then checks if he had received any messages during his training phase, and if it is the case he applies model aggregation for every received message
- The client then starts the training phase again

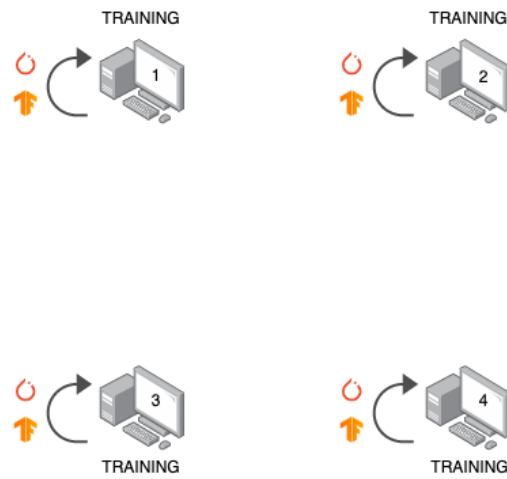
In order to illustrate the different steps of a Asynchronous Decentralized Federated Learning round of training, an example using the same agent network graph as in figure 2.7 is presented in figure 3.1.



(a) The clients train their models locally, using the local training data



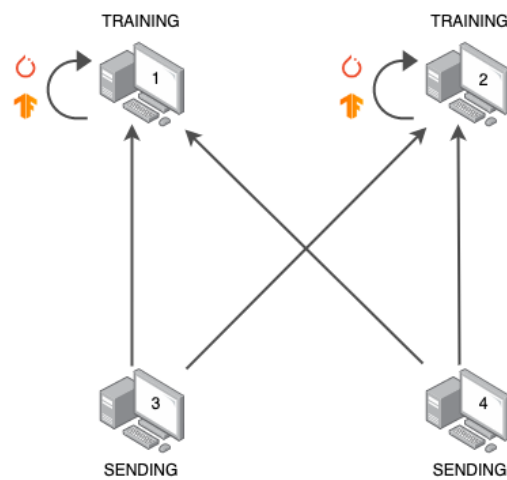
(b) Client 2 is done training, he sends his model weights and biases to his neighbours



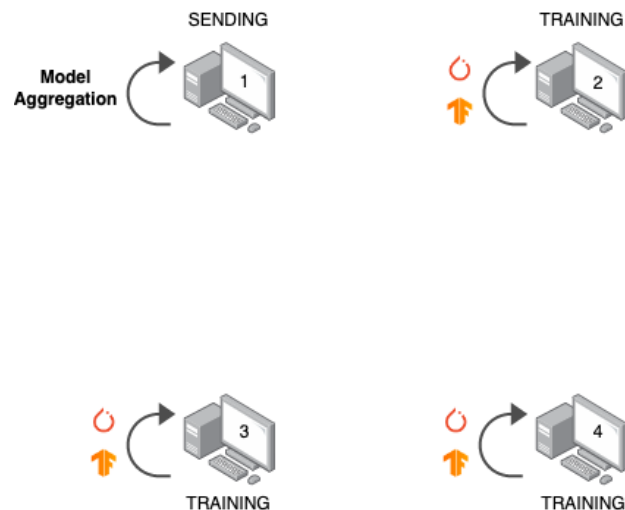
(c) Client 2 resumes training without waiting for answers from its neighbours



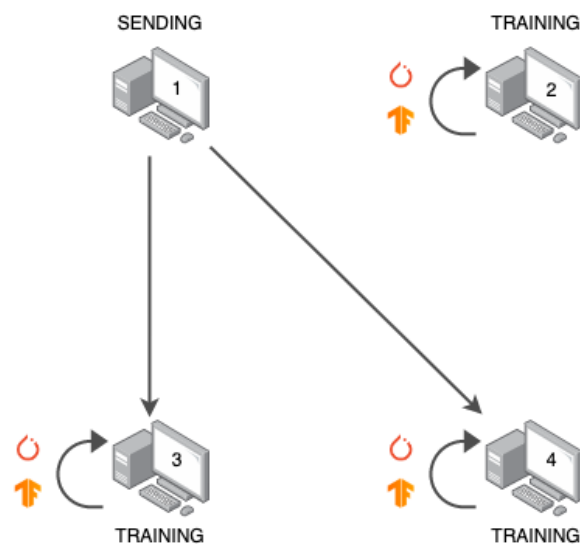
(d) Clients 3 and 4 are done training and apply model aggregation with the previously received weights



(e) Clients 3 and 4 send their model weights to their neighbours



(f) Client 1 is done with training and applies model aggregation with the weights he received during training



(g) Client 1 sends his model weights to his neighbours

**Figure 3.1:** Steps of an Asynchronous Decentralized Federated Learning training round

In this example, we clearly notice that the main difference compared to the example of the Synchronous Decentralized Federated Learning setting (figure 2.8) is that the agents are never waiting. This might seem like a very obvious way to improve the algorithm using the synchronous setting, but it is not as trivial.

The model aggregation performed in the synchronous architecture can be seen as more *powerful* as it combines all the models of all the neighbours of the agent. The aggregation therefore takes into consideration more data, and should logically output a more precise model. The model aggregation performed in the asynchronous architecture only handles one neighbour at the time, which means that for each model aggregation, the model evolves less than in the synchronous setting. The consensus algorithm will therefore have to be applied many more times in the asynchronous setting before reaching similar performances, which might imply increasing slightly the execution times.

### 3.2.2. Asynchronous consensus

The general consensus algorithm described in section 2.3.1, and more particularly the equation 2.9, needs to be slightly adapted to the asynchronous setting. In the setting of this study, the agents communicate between themselves in an asynchronous manner, and the agent network graph evolves dynamically since agents can join or leave the network during the execution. Therefore, the algorithm presented in 2.9 has to be slightly adapted in order to take these factors into consideration. As mentioned previously,  $0 < \epsilon < 1/\Delta$  is a necessary condition to fulfill in order to obtain a consensus. The practical issue with this is that in the implemented MAS, the agents do not have a knowledge of the full network but only of their direct neighbours. Therefore, agents can not know what the maximal degree of the graph is, and can therefore not know what value of  $\epsilon$  to use to perform the consensus. Each agent only knows about the degree of its own node. Thus, agents must communicate their local degrees to each other in order for each agent to be able to determine the maximal degree of the graph. It is important to remember that the maximal degree of the graph is also a dynamic value, as adding or removing agents throughout the execution influences this value.

Since the  $\epsilon$  value used by each agent is a dynamic value, we will call  $\epsilon_i(k)$  the  $\epsilon$  value of node  $i$  after  $k$  iterations. As mentioned previously, the model aggregation phase (during which the consensus algorithm is executed) is applied only with one neighbour at the time. This neighbour, along with its model weights, sends his own local  $\epsilon$  value. Since the objective is to respect the  $0 < \epsilon < 1/\Delta$  condition, a technique would be to always try to minimize the value of  $\epsilon$  in order to make sure it satisfies the constraint. Therefore, every time a message is received, if the  $\epsilon$  value of the neighbour is lower than the  $\epsilon$  value of the agent, then the agent must replace its  $\epsilon$  value by its neighbour's  $\epsilon$  value. Formally, at each consensus step of an agent  $i$ , where a neighbour  $j \in N_i$  sends his step-size  $\epsilon_j(k)$ ,  $\epsilon_i$  is updated as follows :

$$\epsilon_i(k+1) = \min(\epsilon_i(k), \epsilon_j(k)) \quad (3.1)$$

From now on, we will call  $\epsilon(k)$  the real maximal degree of the graph at iteration  $k$ , whereas  $\epsilon_i(k)$  represents the belief of agent  $i$  to what is the maximal degree of the graph.

Since the  $\epsilon$  value is not constant and is constantly updated, the theorem stated in section 2.3.1 does not hold anymore. However, it is understandable that, if the graph  $G$  is connected, then a consensus is asymptotically reached for all agent's initial states. Indeed, when there are no new agents entering the network or agents leaving the network for a reasonable amount of time, the  $\epsilon$  value propagates in the network, all the agents end up fulfilling the condition  $\epsilon_i = \epsilon$  and the problem becomes the one stated in equation 2.9 and the theorem becomes applicable. We can therefore establish that, if there is a certain iteration  $k^*$  such as  $\forall k > k^*$  no more agents enter or quit the agent network graph at iteration  $k$ , then a consensus will asymptotically be reached.

However, there is no proof that the final value of each agent would be the average of all agents' initial values. Indeed, the reasoning that we just made applies to guarantee that a consensus will be reached, but does not say anything about the final value of each agent.

The full proof will not be shown here, but is available in [57], and shows us that the final asynchronous consensus algorithm equation is :

$$x_i(k+1) = \epsilon_i \sum_{j \in N_i} (x_j(k) - x_i(k)) - \left(1 - \frac{\epsilon_i(k+1)}{\epsilon_i(k)}\right) (x_i(k) - x_i(0)) \quad (3.2)$$

The second term of this expression is a correction term, that is added only to be able to ensure that the consensus will be reached. Indeed, it is proven that this term maintains the necessary conditions to ensure convergence [57]. This correction term involves the values of  $\epsilon_i(k+1)$  which is the value of  $\epsilon_i$  used at iteration  $k+1$ , and  $\epsilon_i$  which is the value of  $\epsilon_i$  used at iteration  $k$ . As we see, if  $\epsilon_i(k) = \epsilon_i(k+1)$  then the fractions equals to 1 and the correction term equals to 0, which leads back to the original version of the consensus algorithm 2.9. Therefore, this term only plays a role when  $\epsilon_i(k) \neq \epsilon_i(k+1)$ , which concretely speaking is the moment where the local value of the step-size known by an agent changes.

Furthermore, it can be proven that this algorithm reaches consensus and converges towards the average of the initial values. A sufficient condition to ensure this is that the sum of the values of all clients at every iteration is constant [57]. Indeed, if the sum of all values is constant, the average is constant too, and it is therefore equal to the average of all initial values. The proof that the sum of all values remains constant in this setting is given in [12]. We can therefore ensure that the asynchronous consensus algorithm converges to the average of the initial values, as long as  $0 < \epsilon < 1/\Delta$ .

Equation 3.2 gives a general expression of the asynchronous consensus, but in our case, as mentioned in equation 3.1, the consensus is always applied to only 2 agents at the time. Therefore, the final asynchronous consensus expression can be rewritten as :

$$x_i(k+1) = (1 - \epsilon_i(k+1))x_i(k) + \epsilon_i(k+1)x_j(k) - \left(1 - \frac{\epsilon_i(k+1)}{\epsilon_i(k)}\right) (x_i(k) - x_i(0)) \quad (3.3)$$

Equation 3.3 is therefore the final formula that will be used throughout this work to apply the asynchronous consensus.

### 3.2.3. Algorithm

The ADFL algorithm involves parallelism. Indeed, agents are able to train and receive or send messages simultaneously. Therefore, specifying a single written algorithm in a simple manner is quite difficult. Therefore, we first of all present the main lines of the algorithm that handles the training, the sending of the message and the receiving of the response before applying the consensus (Algorithm 3.1). Then, we present briefly the algorithm of the message reception, and how messages are handled upon reception (Algorithm 3.2). Please note that these are not the fully detailed algorithms, the more specific details will be discussed in section 3.7.1 where the implemented agents' architectures are described. In these algorithms, we denote as  $W_i(k)$  the weights and biases of the model of client  $i$  at iteration  $k$ .

---



---

**Algorithm 3.1** Model training, message sending, response receiving and consensus in the Asynchronous Decentralized Federated Learning algorithm

---



---

```

1: while !killed do
2:    $W_i(k) \leftarrow TRAIN()$ 
3:    $j \leftarrow RANDOM\_SELECT(N_i)$ 
4:    $SEND(W_i(k), \epsilon_i, j)$ 
5:    $W_j(k) \leftarrow RECEIVE\_RESPONSE(j)$ 
6:    $W_i(k+1) \leftarrow CONSENSUS(W_i(k), W_j(k), W_i(0), \epsilon_i(k), \epsilon_i(k+1))$ 
7: end while

```

---



---

This first algorithm shows that when a user finished to train its model, it selects a random neighbour and sends him his weights along with its  $\epsilon$  value. Then, he receives the weights of the neighbour in response to this message. He then applies the consensus algorithm in order to update its weights, and starts training again.

---



---

**Algorithm 3.2** Message receiving and consensus in Asynchronous Decentralized Federated Learning algorithm

---



---

```

1: while !killed do
2:    $\epsilon_j, W_j(k) \leftarrow WAIT\_FOR\_MESSAGE()$ 
3:    $RESPOND(W_i(k), j)$ 
4:   if !training then
5:      $W_i(k+1) \leftarrow CONSENSUS(W_i(k), W_j(k), W_i(0), \epsilon_i(k), \epsilon_i(k+1))$ 
6:   else
7:      $STORE\_WEIGHTS(W_j)$ 
8:   end if
9:   if  $\epsilon_j < \epsilon_i$  then
10:     $\epsilon_i \leftarrow \epsilon_j$ 
11:  end if
12: end while

```

---



---

This second algorithm shows that the agent is always waiting for a message. When he receives the weights of a neighbour, he responds with his own weights, and then applies the consensus algorithm. There is a slight particularity in this moment, as the agent can not perform model aggregation if the agent's model is currently training. If the agent is training, the received weights are simply stored, and will be used later on. This issue will be further discussed in section 3.7.1.

Before implementing the code of the actual application, this algorithm was tested by simulations on Google Colab<sup>1</sup>. In these simulations, agents were represented by their adjacency matrices, and were given different initial values. The communication delays were not taken into consideration, but the experiments showed that the algorithm did converge, which meant that we could start implementing it in the application.

---

<sup>1</sup><https://colab.research.google.com>

## 3.3 Datasets

### 3.3.1. MNIST

In order to test the efficiency of the proposed solution and to assert that it is working as desired, datasets need to be used in order to benchmark the performances. The main dataset used throughout this work is the MNIST database (Modified National Institute of Standards and Technology database) [37]. It is a wide dataset of handwritten digits (Figure 3.2), containing 60,000 training samples and 10,000 test samples. The MNIST dataset is widely used by researchers to test various model performances, and therefore a large number of accuracy benchmarks are available online [37]. This allows us to compare our results with the available benchmarks in order to validate some of the obtained results.



Figure 3.2: Example of MNIST dataset samples











The images contained in the MNIST dataset are all in black or white (which means each pixel of the images can be encoded either by a 0 or a 1), and have the same format (28 pixels wide and 28 pixels high).

Since the goal of this study is not to reach an optimal accuracy score for the MNIST dataset prediction but rather test and compare new AI model training techniques, no pre-processing was applied to the samples contained in the dataset.

### 3.3.2. Fashion MNIST

Another dataset was used in this work in order to give a different benchmark, the Fashion MNIST dataset [71]. This dataset contains 60,000 training samples and 10,000 test samples of images of the same size as the MNIST dataset (28 pixels wide and 28 pixels high). The images show articles of clothing taken from Zalando<sup>2</sup> and the goal is to classify them into 10 different classes. These classes as well as examples of images are shown in figure 3.3.

<sup>2</sup>www.zalando.com

Label	Description	Examples
0	T-Shirt/Top	
1	Trouser	
2	Pullover	
3	Dress	
4	Coat	
5	Sandals	
6	Shirt	
7	Sneaker	
8	Bag	
9	Ankle boots	

**Figure 3.3:** The 10 Fashion MNIST classes and examples of dataset samples

As explained previously, no preprocessing was applied to the images contained in this dataset.

This dataset is not used for the experiments presented later in this project, but is useful in order to test that the proposed solution works for multiple different datasets.

In practise, the datasets were imported using the `torchvision` library [44], through the `torchvision.datasets.MNIST` and `torchvision.datasets.FashionMNIST` classes. If it is the first time the code is executed on a device, the code fetches the dataset from a server and saves it locally in a folder named `data` in the root repository. For all the next executions, the dataset will directly be fetched in the local folder to avoid having to download it every time.

It is also important to describe how the agents obtain the data they will train on. The `Sampling.py` file allows to select a random subset of the MNIST or Fashion MNIST dataset. In practise, each agent trains on 300 random images.

Next, it is also important, as mentioned in section 2.2.1, to split the data into a training set and a test set. In the case of this study, the split was made such as 80% of the available data was used for training and 20% for testing. This splitting is done in the `Utilities.py` file.

### 3.4 Models

The principles of Federated Learning apply to any type of models, as long as model information can be sent from one agent to another. The selection of the model is therefore not crucial in the elaboration of the solution presented in this study. In order to have



a clear and simple representation of the results obtained, this work uses mostly quite comprehensive models, in order to illustrate more clearly the principles of Asynchronous Decentralized Federated Learning.

### 3.4.1. Multilayer Perceptrons

In the first place, the models used for testing the solution were Multilayer Perceptrons (MLP). These models belong to the class of feed-forward Neural Networks. Feed-forward Neural Networks are a subcategory of Artificial Neural Networks, also called Neural Networks (NN). These are computing systems that allow a model to predict an output based on data given as input and a training process. A Neural Network is a collection of connected Artificial Neurons. Each neuron can receive signals and send numbers (weights) to the other neurons they are connected to. These weights then allow to compute a mathematical value based on the input values and output a prediction. The training process consists in changing these weights in a way that the prediction outputted by the model is as close as possible to the expected output.

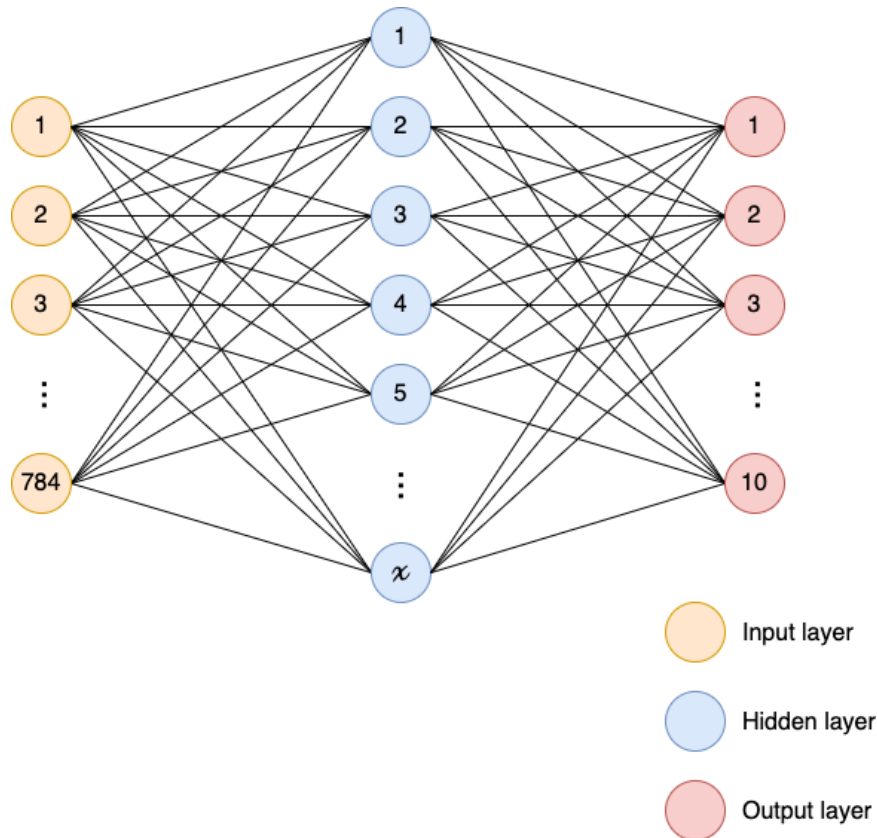
Neurons are typically aggregated into layers, as in the Multi-Layer Perceptron for example. The first layer is called the input layer, where the raw data is given as input. The last layer produces the prediction of the model, it is called the output layer. The layers in between the input and output layer are called hidden layers. Feed-forward Neural Networks are a subset of Artificial Neural Networks where the connections between the nodes do not form a cycle. Therefore, the information moves in only one direction : from the input layer nodes to the hidden layer nodes, until the output nodes.

The MLP models used in this study contain 3 layers :

- the input layer which contains the input data to the Neural Network, the number of neurons it is made out of depends on the size and nature of the input data
- one hidden layer, which consists in an arbitrary number of neurons
- the output layer, which contains as many neurons as there are possible outcomes to the prediction problem

As mentioned in section 3.4, the images have a total size of 28 by 28 pixels, therefore the total number of pixels is 784. Since each input to the MLP corresponds to the value of one pixel, the input layer of the MLP consists of 784 neurons. The goal of the model is to recognize handwritten numbers, going from 0 to 9, therefore there are 10 possible prediction outcomes, which means that the output layer must contain 10 neurons. The number of neurons in the hidden layer remains to be chosen, this is a parameter that will be tuned throughout in order to evaluate its impact on the efficiency of the solution (this parameter is referred to as  $x$  in Figure 3.4).

In practise, these models were created using the Pytorch library [55]. A custom class called MLP was created, which inherits from the `nn.Module` class, which is a Pytorch base class for all Neural Network modules. In the `forward` method defined in this class, the different steps of the training of the model are specified. First, the data enters into the input layer, which is defined using the `nn.Linear` class, which means that the transformation applied to the incoming data will be a linear transformation (of the type  $y = ax + b$ ). Then, the result is passed through a dropout layer. This layer randomly sets some of the weights to 0, in order to avoid an *overfitting* of the model. The result is then passed to the



**Figure 3.4:** Multilayer Perceptron architecture

ReLU activation function [2]. This activation function brings non-linearity to the model and allows the model to solve more difficult problems by increasing its complexity. Then, the result is passed to the hidden layer, which is also defined with the `nn.Module` class. Finally, the result passes through the softmax activation function [50], which allows to convert the final 10 weights into 10 numbers between 0 and 1, corresponding to the probabilities of the sample to belong to each class. In order to obtain the output of the model, it therefore suffices to identify which output neuron displays the maximal probability. For example in the case of MNIST, if the 3rd neuron in the output layer has the maximal output out of all the neurons of the layer, the prediction of the model is that the input image is a 2.

### 3.4.2. Convolutional Neural Networks

The other models that were used in this solution are Convolutional Neural Networks (CNN). These models also belong, just like MLPs, to the class of Artificial Neural Networks, and are particularly used to analyze visual imagery [67]. In opposition to MLPs where the image is flattened and each input to the model corresponds to one pixel value of the image, CNNs are able to capture spatial dependencies on the picture. Indeed, when an image is flattened (when the matrix of pixel values is transformed into a unidimensional vector), the spatial dependency is completely lost, which is avoided by the use of CNNs. The role of the CNN is to reduce the images into a form that is easier to compute for the models, and keep all the important features of these images. This first of all allows the model developer to perform way less extensive pre-processing, and also saves a lot of execution time as the input data is smaller.

CNNs are composed of 3 types of layers : Convolutional Layers, Pooling Layers and Fully-connected Layers [52]. The first part of a CNN model is called the Feature Learning, in which the Convolutional Layers and Pooling Layers are used to extract features and compress the input image. In the second part, the Classification part, the Fully-connected Layers take as input the output of the Feature Learning part, and work exactly the same as a traditional Artificial Neural Network, in order to produce an output. An example of this architecture is shown in figure 3.5, where the layers in blue represent the Feature Learning part and the 2 Fully-connected layers represent the classification part.

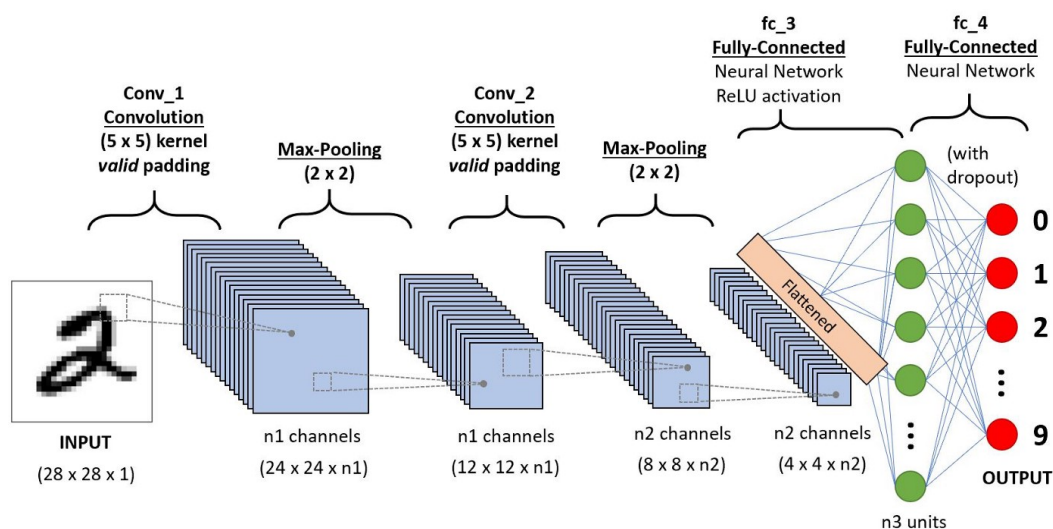
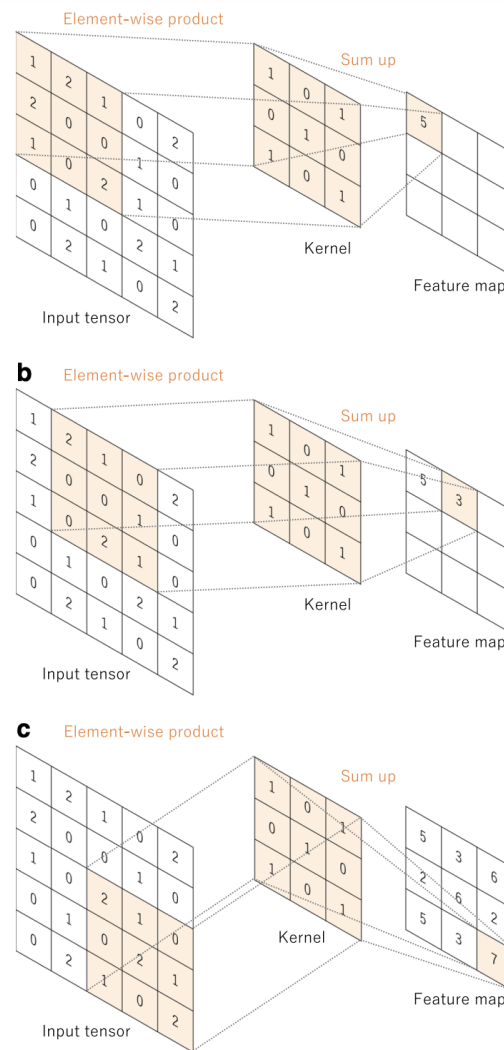


Figure 3.5: Convolutional Neural Network architecture

## Convolutional Layer

The Convolutional Layer consists in a combination of linear operations (the convolution) and non linear operations (the activation) [72].

The convolution operation's goal is to extract features from the input image. In order to do this, a small matrix of numbers, called *kernel*, is applied to the input image, such as for each pixel of the image an output value is computed based on the kernel and on the neighbouring pixels of the image. This computation is done by an element-wise operation between each element of the kernel and each corresponding element of the input image, and by summing all these values to obtain a single output value for one input pixel. The output is therefore a matrix of numbers, and is called a *feature map*. An example of convolution is showed at Figure 3.6. We can see that the kernel does not change, but it is simply placed at each possible position on the input image.



**Figure 3.6:** Example of 3 convolution operations. This figure was taken from [72]

On the previous example, we can notice that the obtained feature map has a smaller dimension than the input image. This is due to the fact that the pixels on the border of the image can not be computed because they do not have enough neighbouring pixels to use the kernel. In some cases, adding pixels around the border of the image (called *padding*) can be useful in order to obtain a feature map of the same size as the input image. One solution is to apply *zero-padding* which means that pixels with a value of 0 are added all around the borders of the input image in order to have a feature map of the same size as the input image. The distance which the kernel moves between 2 successive convolutions is called the *stride*, it is equal to 1 on the example in figure 3.6. Multiple kernels are used and therefore multiple feature maps are produced at each round.

Once the feature maps are produced, they are passed through a non-linear activation function in order to bring non-linearity to the model. The most common activation function used in the case of CNNs is ReLU (Rectified Linear Unit) [56].

## Pooling Layer

The Pooling Layer performs a downsampling operation on the input that it is given. Indeed, it allows to reduce the size of the feature map, and this makes the model more robust to small translations and shifts in the input image. There are multiple ways of performing pooling, but the most common one is max-pooling.

Max-pooling consists in keeping only the maximum value of a group of values in a feature map and outputting it. The most common max-pooling is performed with a  $2 \times 2$  kernel, which means that for each square of 4 values in the feature map, only the maximal one is kept. This, of course, reduces the dimensions of the feature map by a factor of 2. An example of max-pooling with a kernel of  $2 \times 2$  is presented at figure 3.7.

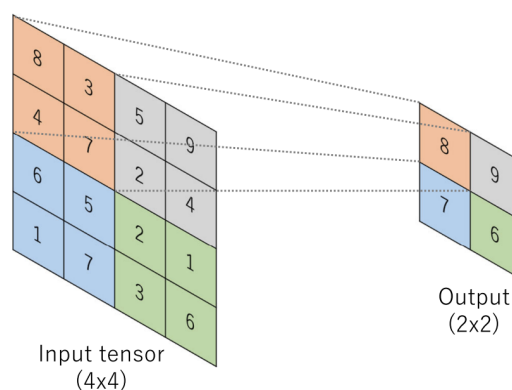


Figure 3.7: Example of a max pooling with a  $2 \times 2$  kernel. This figure was taken from [72]

## Fully-Connected Layer

After a series of convolutions and pooling steps, the data is completely flattened, which means a 2 dimensional image is transformed into a unidimensional array, and given as input to an Artificial Neural Network, which works just as the MLP as explained in section 3.4.1. The input to this Neural Network is therefore not image pixels as previously, but rather numbers that correspond to features identified in the image through the convolutions and pooling steps.

In practise, the CNN used in this study has the following components :

- A Convolutional Layer that produces 10 feature maps in output, using a kernel of size  $5 \times 5$ .
- A Max-Pooling layer using a kernel of size  $2 \times 2$ , and applying the ReLU activation function on the output.
- A second Convolutional Layer that produces 20 feature maps in output, using a kernel of size  $5 \times 5$ .
- A Dropout layer that randomly sets some values to 0 in order to avoid an overfitting of the model.
- A second Max-Pooling layer using a kernel of size  $2 \times 2$ , and applying the ReLU activation function on the output.

- A Fully-Connected linear input layer, which has 320 neurons, after flattening the data. It then applies the ReLU activation function to the data.
- A second Dropout layer that randomly sets some values to 0 in order to avoid an overfitting of the model.
- A Fully-Connected linear hidden layer that contains 50 neurons.
- An output layer of 10 neurons.

### 3.5 Model Training

---

In order to train the model, the Pytorch library was used. The `train_local_model` function of the `Federated.py` file is called in order to perform one training round of the model and return all the results of the training. The optimizer used in order to train the model and perform backtracking is SGD (Stochastic Gradient Descent). The images of the dataset are regrouped in batches of 3 images, and the model is trained on all batches once. The loss criterion used in the training is the Negative Log Likelihood Loss (NLL) implemented in Pytorch.

Next, the model is saved locally by default so that it could later be used as an input when executing the code. Indeed, the file containing the model weights could be used to define the initial weights of a model when launching the program.

Once the training and saving of the model is complete, the accuracy and loss of the training round is computed and returned to the agent.

### 3.6 Consensus

---

As explained in 3.2.2, the consensus algorithm in this solution is applied only by 2 agents at the time, following equation 3.3. In practise, this is done in the `Consensus.py` file, using the `apply_consensus` function. This function takes as input the weights of the agent the weights of the neighbour with whom the consensus has to be applied with, and applies the consensus formula for each layer of the model successively.

The arrays of weights of each layer have to be flattened in order to be able to manipulate them and apply the consensus algorithm. At the end of the process, the results are formatted in the same way as the input weights, and the final weights are returned to the agent, which updates its model with these new weights.

### 3.7 Agents

---

As mentioned previously in sections 2.1.1 and 2.1.2, agents are entities that are placed in an environment. In the case of this study, the environment corresponds to the knowledge of a part of the agent network graph. Indeed, all agents need to know with which particular agents they need to connect in order to exchange information. The features of the developed environment are described in table 3.2. This environment is fully accessible

since an agent always knows to who he must send his messages to. Furthermore, it is a fully deterministic environment as the agents modify the environment in a deterministic way. The only situations when the agents can bring changes to the environment are when they connect or disconnect, which effectively modifies the agent network graph in a deterministic fashion. The environment is also dynamic, since new agents can join at anytime, and therefore modify the agent network graph. Finally, the environment is discrete since an agent can only be in a finite amount of states, the interaction with the environment is not defined by a continuous function.

<b>Environment Feature</b>	<b>Implementation</b>
<b>Accessibility</b>	Fully Accessible
<b>Determinism</b>	Fully Deterministic
<b>Dynamism</b>	Dynamic
<b>Continuity</b>	Discrete

**Table 3.2:** Characteristics of the MAS environment implemented in the solution

The clients in the FL setting are each represented by one SPADE agent, represented by the FLAgent class. As stated previously, agents contain various behaviours that allow it to perform the desired tasks. Furthermore, a well-designed and user-friendly web interface should be created for each agent in order to be able to track relevant data during the execution (for example the accuracy after the last training round, the history of messages received, ...).

### 3.7.1. Agent architecture

The agents built in this solution contain 3 behaviours :

- A Presence Behaviour which handles the notification of presence, which allows agents to notify their neighbours when they connect or disconnect.
- A Message Receiving Behaviour, which allows agents to constantly be ready to receive messages and apply the consensus algorithm in an asynchronous fashion.
- A State Machine Behaviour which allows to transition between the training, sending and receiving states in order to follow the Asynchronous Decentralized Federated Learning scheme.

#### Presence Management

As soon as an agent connects, the Presence Behaviour automatically identifies its neighbours in the graph that as given as an input to the agent, and subscribes to all of them. If the neighbours are connected, they will therefore receive a notification stating that this agent is trying to subscribe.

When an agent receives a subscription request, he accepts it, then, if he is not already subscribed to this agent, subscribes back. All agents keep track of the active neighbours



that they are subscribed to in a list, and keep updating this list when a new agent subscribes or disconnects.

When an agent disconnects (he becomes *unavailable*), the presence mechanism implemented in SPADE allows to notify all subscribed agents about this event. Therefore, when this occurs, the agent simply removes the disconnected agent from its list of active neighbours.

Furthermore, as mentioned in 2.3.1, the maximal degree of the graph is an essential piece of data in order to apply the consensus algorithm. However, since the graph is dynamic as agents can connect and disconnect whenever they want to, this maximal order is also a dynamic value. Therefore, it needs to be updated everytime a new agent connects or disconnects. The Presence Behaviour allows to constantly keep track of the number of active neighbours the agent has, and it can therefore update the maximum order of the graph when new agents are added or removed. The maximal degree known by the agent is saved locally, and is sent along with the weights of the agent everytime the agent sends its weights.

### Receiving Messages

A behaviour dedicated to constantly receive messages was implemented. This behaviour waits for a message to arrive from one of the agent's neighbours, then responds to this agent with its own weights and finally applies the consensus using the model weights and biases that were contained in this message. The message also contains the maximal degree of the graph to the knowledge of the neighbour agent. Therefore, the agent checks if the neighbour's  $\epsilon$  value is inferior to his own, and if it is the case he replaces his own value by the one of the neighbour. Note that, in accordance with the asynchronous character of the implementation, this behaviour is constantly running in parallel to the other behaviours of the SPADE agent.

### State Machine

The State Machine Behaviour is a behaviour that contains several states, and allows to transition from one state to another based on certain conditions. In this case, 4 different states are needed in order to build the solution :

- The **SETUP** state : the initial state of the State Machine, can be used to perform operations before starting the other processes
- The **TRAINING** state : performs the training of the local model
- The **SEND** state : after the training of the local model, sends the model's weights and biases to a randomly chosen neighbour of the agent
- The **RECEIVE** state : waits for a response to the message sent in the **SEND** state and applies the consensus algorithm when the weights of the other agent are received

Note that sending the weights is not as trivial as it might seem. Indeed, the weights are tensors from the Pytorch library, and the SPADE library only allows to send strings (characters). Therefore, the model weights and biases have to be converted to a string



before being sent, which is done using the `pickle` and `codecs` libraries. Also, as mentioned previously, the maximal degree of the graph to the knowledge of the agent has to be sent along these model weights. Therefore, a "|" symbol is added after the string encoding of the weights and biases, and the maximal degree is then added to the end of the message before sending it. Once the message is received, it has to be decoded using the same libraries in order to transform the string back to a Pytorch tensor and get the maximal degree of the graph to the knowledge of the sender of the message as well as his weights.

### Global running of an agent

Now that the 3 behaviours used in the SPADE agents have been defined, it is important to understand how they interact between themselves in order to follow the Asynchronous Decentralized Federated Learning scheme.

First of all, when an agent is executed but none of his neighbours are connected, he can of course not send his model's weights and biases. In this case, instead of waiting, the agent trains locally its model until at least one of its neighbours connects.

In the case of the State Machine, the execution is pretty straightforward : the agents trains its model locally in the TRAINING state, then switches to the SEND state in order to send its model's weights and biases, and then finally to the RECEIVE state in order to receive a response from the neighbour he sent the weights to, and applies the consensus algorithm. However, all this process implies that the agent did not receive any message from one of his neighbours asking for his weights throughout the execution. Indeed, if a neighbour asks the agent's model's weights and biases during the training phase, the question is to know with which weights to apply the consensus. If the consensus is made with the incoming weights from the neighbour and the pre-training weights of the agent, then the whole training process that has been made during the current round is lost. Indeed, the training of a model consists in changing the model's weights, so not taking these changes into consideration would represent a huge waste of time and resources.

The first solution to this problem was to interrupt the training phase if a message arrives from one of the agent's neighbours during the training phase, apply the consensus with the weights of the model before the training phase, replace the mid-training weights by the weights obtained by the consensus, and then resume the training. This solution is represented in figure 3.8.

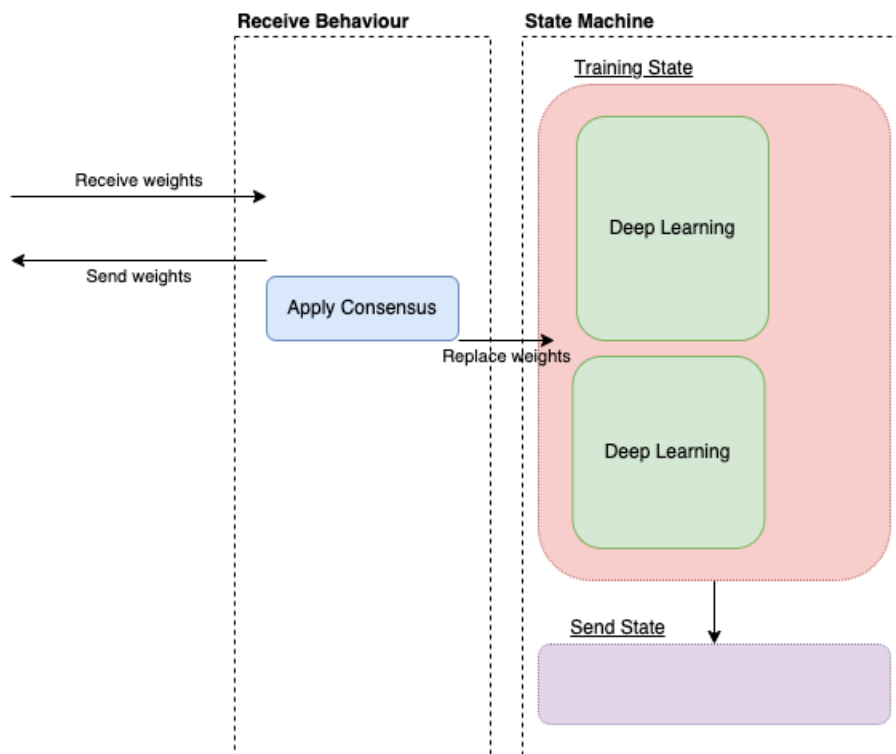


Figure 3.8: First proposed solution

The problem with this solution is that the first part of the training (before the interruption) is still lost, as the weights obtained through the consensus replace the weights of the current training round. Therefore, the modifications made to the weights are lost in the process, and this part of the training proves to be useless.

A second solution was therefore implemented, which is presented on figure 3.9. In this solution, the training part is never interrupted. When a message arrives from a neighbour, the agent switches the value of a flag (here called *weights\_received*) to 1. At the end of the training phase, the agent systematically verifies the value of this flag, and if it is set to 1, applies the consensus on the model that is resulting from this training phase. The flag is then of course reset to 0. This solution allows to never waste any training time, which improves the efficiency of the solution.

There is however a slight twitch to me made to this solution, as we only considered the situation where one neighbour sends a message to the agent, whereas there could be multiple neighbours sending messages to the agent during one training phase. Therefore, instead of using a flag as in figure 3.9, a queue was implemented to store all the messages that arrived during a training phase. After the training round, the agent goes through all the messages that were stored and applies the consensus algorithm one by one with all of them. This solution is illustrated at figure 3.10, which is the final solution adopted in this study.

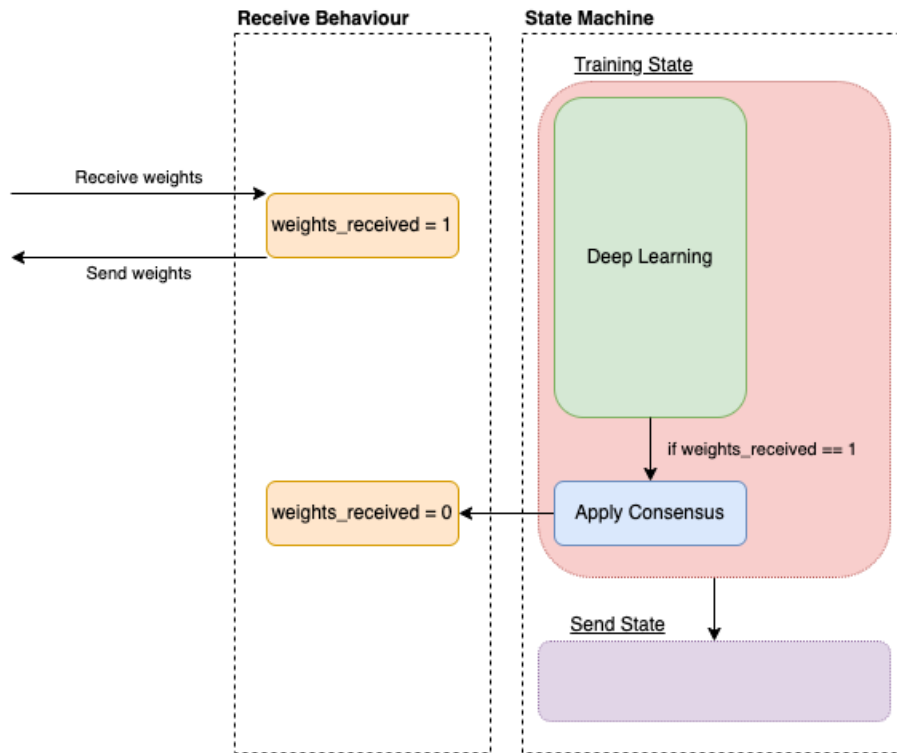


Figure 3.9: Second proposed solution

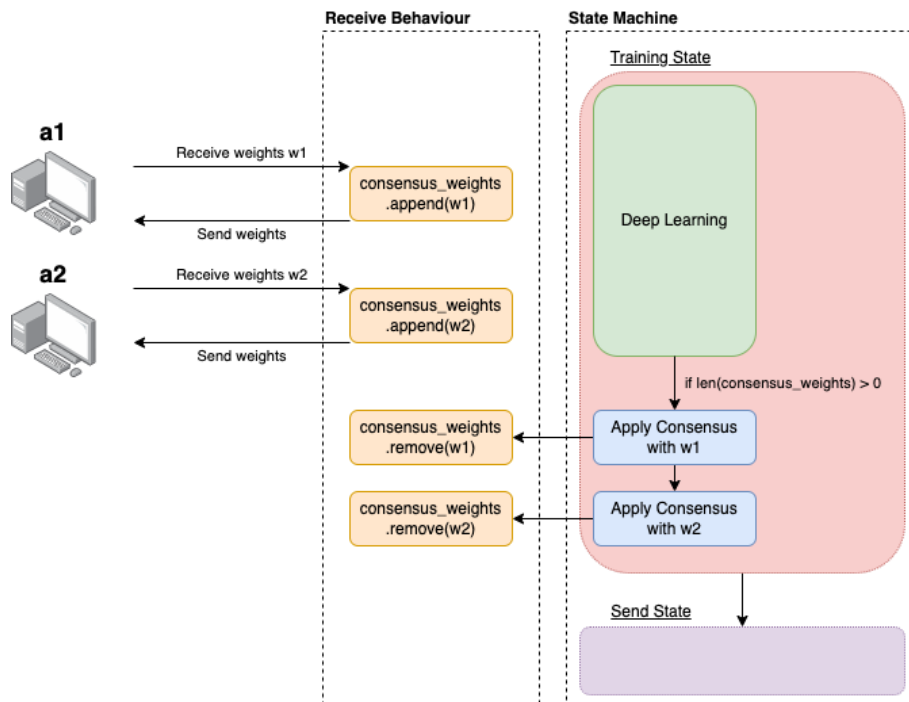


Figure 3.10: Third proposed solution

### 3.7.2. Agent Web Interface

As mentioned in section 2.1.4, the agents can have default graphical interfaces for the user to be able to get some information on the execution. We also discussed that custom graphical interfaces could be created. In this study, an objective is that the user of the application should have a clear understanding of the execution of the program, and have clear metrics to evaluate the performance of the algorithm during the execution. Therefore, a custom graphical interface was developed for the agents showing several information about the ongoing execution, it is presented in figure 3.11.

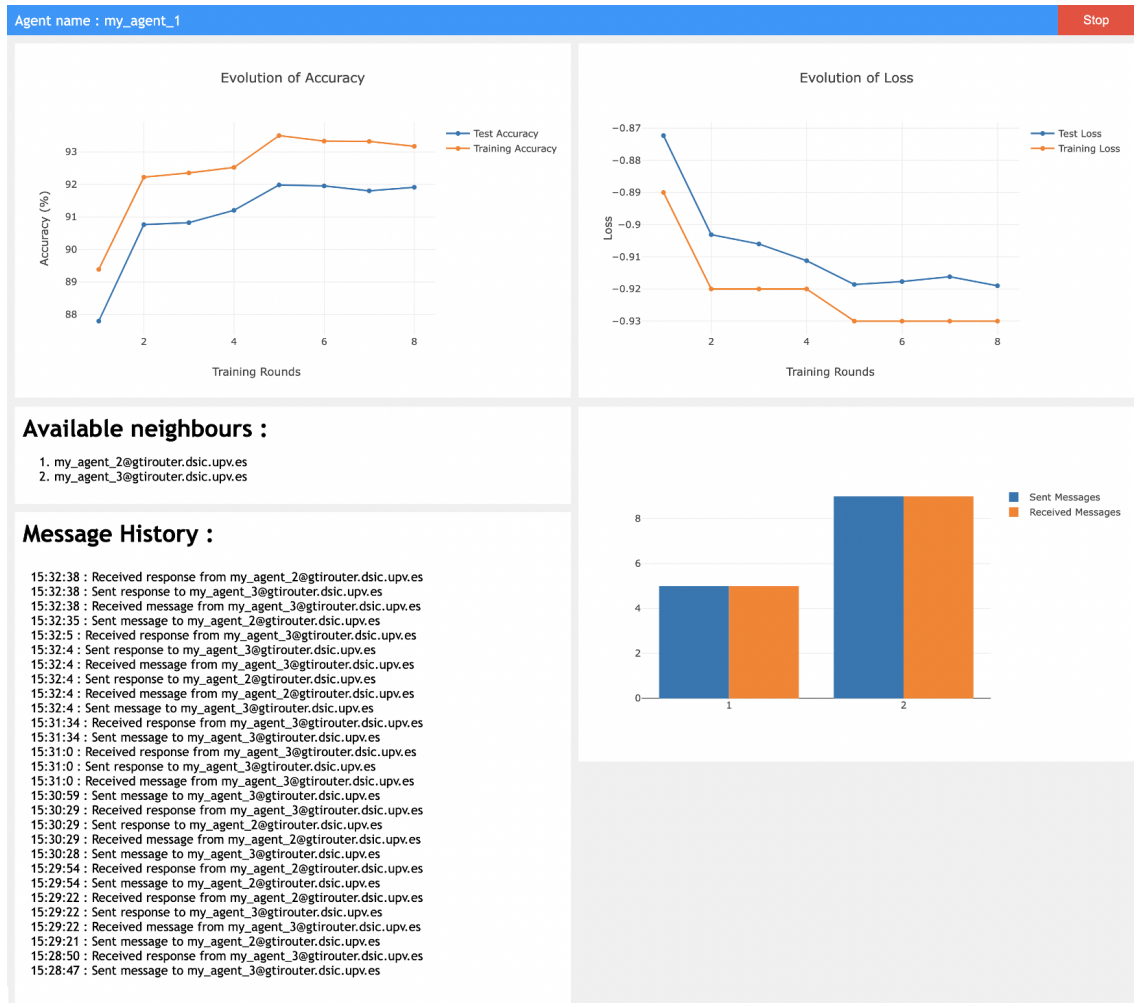


Figure 3.11: Agent's graphical interface

As we can see, the top banner of the page shows the name of the agent, along with a STOP button on the right. This button allows to kill the agent by simply clicking it. The training, messaging, and all other processed carried out by the agent will then simply stop.

Next, 2 graphs are presented. The first one displays the evolution of training accuracy and test accuracy in function of the number of training rounds that have been completed. The second one shows the evolution of the training loss and test loss in function of the completed training rounds. These 2 graphs are interactive, the user can

change the scales, move them around,... These graphs were built using the Plotly<sup>3</sup> library for JavaScript. Indeed, although the code is in Python, the interface is coded in HTML and some inline JavaScript code has been implemented, for example to create the several graphs.

Then, a list of all available neighbours of the agent is shown. Below that, a history of all past sent and received messages is shown. For each received message, the sender is indicated and for each sent message the receiver is shown.

Finally, the graphical web interface displays a bar plot which indicates the number of sent and received messages in function of the neighbour. Indeed, the blue bar represents the amount of messages the agent has sent to this agent, whereas the orange bar represents the amount of messages the agent has received from this agent. The number below each bar refers to the number of the agent in the list of available neighbours.

The content and appearance of this graphical interface is defined in the `agent.html` file, located in the directory `Agents/Interfaces`. In practise, the SPADE agent contains a function called `agent_web_controller`, which sends various data to the graphical interface periodically. For instance, the data sent is the evolution of the training and test accuracy and loss, the names of the neighbours of the agent,... This function also allows to format some elements before sending them to the interface. For example the text corresponding to the message history is built in this function, as well as the arrays containing the number of sent and received messages per neighbour.

### 3.7.3. Launcher Agent

In order to have a more complete solution and offer an optimal user experience, another SPADE agent was created, whose goal is to offer a clear interface for the user to setup the agent before launching it. Therefore, this launcher agent is the first one to be launched during the execution. It collects the input from the user and then creates another SPADE agent as explained previously. This agent was created in the `LauncherAgent` class.

The first thing that is displayed on the interface of this launcher agent is a choice between 2 options : creating an agent based on an network graph, or creating an agent without giving a network graph as input. Indeed, if the user gives the agent network graph as input and precises which agent of this network he wants to launch, the neighbours of this agent can easily be identified thanks to the graph. The input format of the network agent graph should follow the GraphML<sup>4</sup> format. This format allows to describe graphs using XML. Each node of the graph has its own identifier, and all edges are listed in the GraphML file, whether they are directed or undirected.

In the case that the user wants to give an agent network graph as input, the different pieces of information that the user can give as an input to setup the agent are :

- the GraphML file corresponding to the agent network graph
- the identifier of the agent in this previously uploaded graph

---

<sup>3</sup><https://plotly.com/>

<sup>4</sup><http://graphml.graphdrawing.org>

- the port on which the agent's graphical interface will be accessible
- a file containing the initial weights that the model should have before starting training (optional)
- a choice to use either the MNIST or Fashion MNIST dataset
- a choice to use either a MLP or a CNN as a model

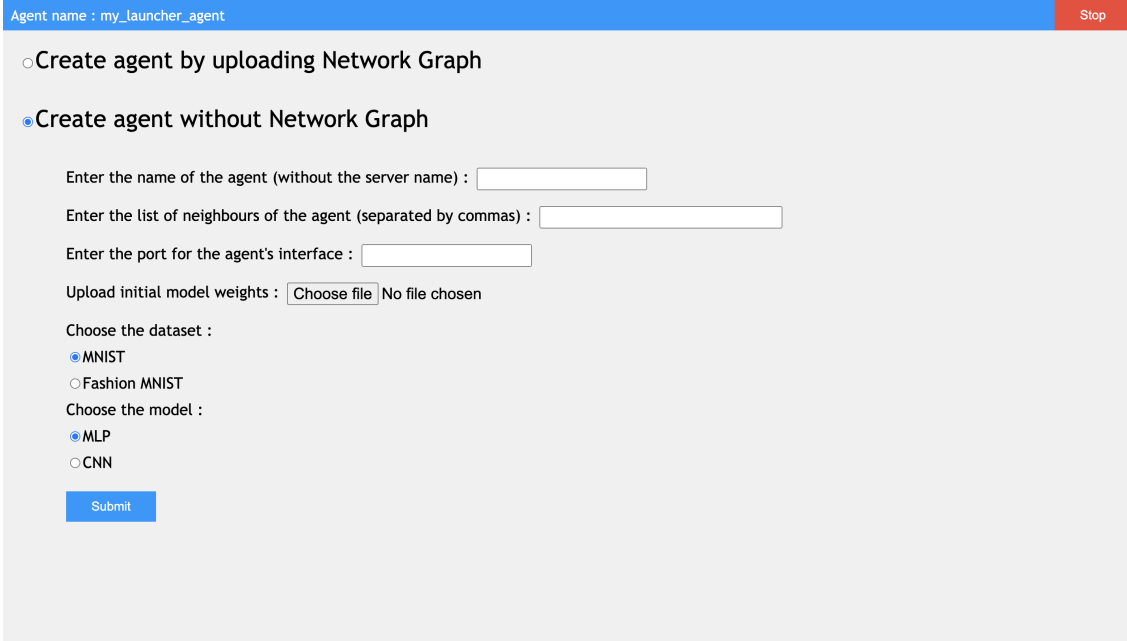
The graphical interface presented to the user in order for him to enter this data is presented in figure 3.12.

**Figure 3.12:** Graphical Interface when the user selects the option to create the agent by uploading a Network Graph

In the case where the user does not want to input an agent network graph to create the agent, he must specify the following :

- the name of the agent, without specifying the server name as this is a default parameter stored in the code
- the list of neighbours of the agent (also without specifying the name of the server) separated by commas
- the port on which the agent's graphical interface will be accessible
- a file containing the initial weights that the model should have before starting training (optional)
- a choice to use either the MNIST or Fashion MNIST dataset
- a choice to use either a MLP or a CNN as a model

In the case where the user does not wish to specify a network graph as input, the graphical interface is as presented in figure 3.13.



The screenshot shows a web-based graphical interface for creating an agent. At the top, a blue header bar displays 'Agent name : my\_launcher\_agent' on the left and a red 'Stop' button on the right. Below the header, there are two radio button options: 'Create agent by uploading Network Graph' (unselected) and 'Create agent without Network Graph' (selected). Under the selected option, there are several input fields and a file upload button: 'Enter the name of the agent (without the server name) :', 'Enter the list of neighbours of the agent (separated by commas) :', 'Enter the port for the agent's interface :', and 'Upload initial model weights : Choose file | No file chosen'. Below these are two sections for selection: 'Choose the dataset :' with radio buttons for 'MNIST' (selected) and 'Fashion MNIST', and 'Choose the model :' with radio buttons for 'MLP' (selected) and 'CNN'. At the bottom left of the form area is a blue 'Submit' button.

**Figure 3.13:** Graphical Interface when the user selects the option to create the agent without a Network Graph

We also notice that in both interfaces, there is a `Stop` button at the top right that allows to kill the launcher agent at any time.

Once the user submits the information, the launcher agents collect the data and create the training agent, specifying the following parameters :

- the Jabber ID of the agent (name and server name)
- the password of the agent (set to a default value in the code)
- the port on which the agent's graphical interface will be available
- the dataset that will be used
- the type of model that will be used
- the list of neighbours of the agent
- the path of the file of the initial weights of the agent's model (optional)

## 3.8 Logging

---

In order to track the execution of the code and evaluate if the solution works as desired, several logging systems were implemented. These logs allow not only to control that the execution is working as expected, but also allows to create graphs and other visual products about the executions. The different logging systems that were created are :

- **Message Logs** : these logging files contain the history of all sent or received messages by the agent. Each line of this file contains the exact time when the message was sent or received, the ID of the message and the sender/receiver of the message. This allows to verify that the messaging scheme is respected. An example of a Message Log file is presented in appendix [A.1](#).

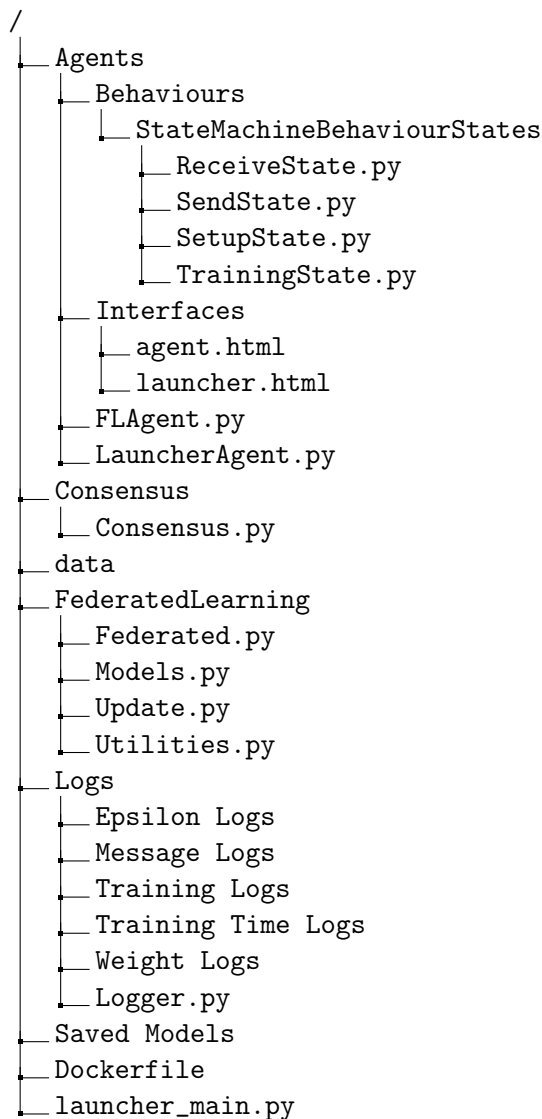
- Training Logs : these logging files contain data about model training. For each round of training, this file contains the exact time, the training accuracy, the test accuracy, the training loss and the test loss. This allows to evaluate the performance of the model and of the overall solution. An example of a Training Log file is presented in appendix [A.2](#).
- Weight Logs: these logging files contain the values of certain weights of the agent after each round of training or whenever consensus is applied. This allows to monitor if the consensus algorithm works effectively. An example of a Weight Log file is presented in appendix [A.3](#).
- Epsilon Logs : these logging files contain data about the  $\epsilon$  parameter. Indeed, as mentioned previously this is a dynamic parameter and it has to be exchanged between the agents of the network. Every time the local  $\epsilon$  value of an agent changes, it is reported in this file. An example of a Epsilon Log file is presented in appendix [A.4](#).
- Training Time Logs : these logging files simply contain data about when the model training phase starts and stops. This allows to have a clear idea of how long the model training process is, and to evaluate which parameters could influence this execution time. An example of a Training Time Log file is presented in appendix [A.5](#).

### 3.9 Code structure

---

Several versions of the code of the implementation exist, which each are dedicated to a certain objective. For example, one of the versions also contains a lot of scripts in order to produce graphs and histograms from the logs. In the final version of the code, which is the one made available to the user, the file structure is the following :





The repository is divided into 6 parts :

- Agents : Code related to the Agents and Launcher Agents, their behaviours, and their graphical interfaces
- Consensus : Code related to the consensus algorithm.
- data : Folder where Pytorch saves the datasets locally.
- FederatedLearning : Code related to Federated Learning, loading the datasets, creating and training the ML models.
- Logs : Code related to logging.
- Saved Models : Folder where the models are saved after each training round.

Concerning the Agents, a separate file was created for each of the agent's behaviours, and grouped under a folder named Behaviours. The HTML codes used to create their graphical interfaces are grouped under the Interfaces folder.

Concerning the code related to Federated Learning, separate files were created for model creation and initialization, and model training (where the weights of the model

get updated).

The `Dockerfile` file present in the root directory will be detailed in section 3.10.2.

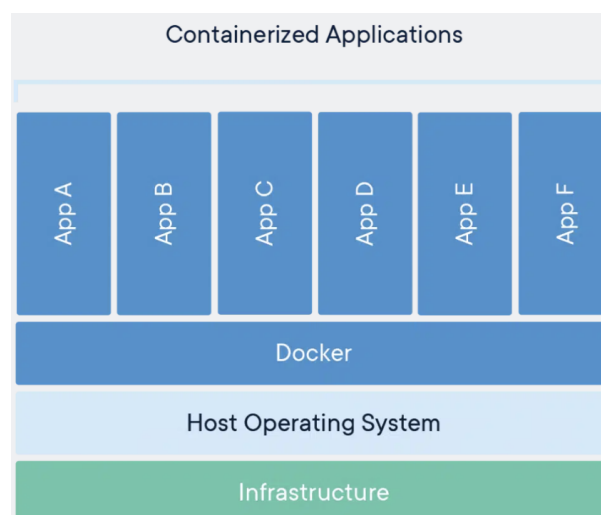
The `launcher_main.py` script in the root repository is the file to execute in order to create the Launcher Agent. As explained previously, this launcher agent will then, after receiving input from the user, create an agent. This is therefore the script to be executed in order to launch the program.

## 3.10 Containerizing the solution

With the objective to present a standalone and easy to use application, having the user to install all the dependencies by himself and launch the `launcher_main.py` script does not seem like an appropriate solution. The last step of the construction of the solution was therefore to find a solution so that the user could extremely easily execute the application. In order to achieve that, the last step of the construction of the solution is to create a Docker image of the program so that it could be executed from a Docker container.

### 3.10.1. Docker

Docker<sup>5</sup> is a platform that allows to run applications in containers. A container is a single piece of software that allows to run an application very easily. Indeed, the container contains the code to run but also all the dependencies, which allows the user to simply execute the container without having to worry about the dependencies or the platform issues. When a Docker container is created, any user on any platform that runs Docker can execute it in the same way and have the same user experience. Figure 3.14 shows that Docker executes a level above the host Operating System, and then contains the different containerized apps.



**Figure 3.14:** How Docker allows to create containerized applications. This figure was taken from <https://www.docker.com/resources/what-container/>

<sup>5</sup><https://www.docker.com/>

In the case of this work, creating a container out of the program seems like a good solution since it avoids all the users to have to install all the dependencies of the code, which can be quite long. Furthermore, there can be issues with the versions of the libraries that have to be installed, and the fact to have a container that already has everything installed is helpful. In addition, the launching of the application becomes much easier and it gives this work an aspect of a finished standalone easy to use solution.

A Docker image is a package of executable software, with all the required dependencies and settings required to run the program correctly. A Docker image is used to create a Docker container which will effectively run the application. Therefore, creating a Docker image is the first step in containerizing a program.

### 3.10.2. Creating a Docker image of the program

The first step before creating the Docker image was to make an exhaustive list of all the dependencies that were necessary to install. This file, called `requirements.txt` was generated automatically by the IDE used in this project (see 6.1.2) and contains all the Python libraries to install along with their version.

The dependencies file generated by the IDE is as follows :

```
1 pandas~=1.4.2
2 matplotlib~=3.5.2
3 spade~=3.2.2
4 termcolor~=1.1.0
5 networkx~=2.8
6 torch~=1.11.0
7 numpy~=1.19.5
8 torchvision~=0.12.0
9 tensorboardX~=2.5
10 scikit-learn~=1.0.2
```

As we can see, there are a lot of libraries used in this project, and it would be long for a user to install all these libraries with the correct versions.

Next, a `Dockerfile` had to be written. This file contains instructions that Docker will execute in order to create the Docker image. The `Dockerfile` used to create the image was :

```
1 FROM ubuntu:20.04 as base
2
3 RUN apt-get update -y
4 RUN apt-get install -y python3-pip
5 RUN pip install --upgrade pip
6
7 COPY requirements.txt /usr/src/
8 RUN pip install -r /usr/src/requirements.txt
9
10 COPY . /usr/src/
11 WORKDIR ./user/src
12
13 ENTRYPOINT ["python3", "/usr/src/launcher_main.py"]
```

Line 1 indicates that our solution will run on Ubuntu, a Linux distribution, on the 20.04 version. Initially, Alpine was the choice of distribution but it gave place to errors

while creating the image, which is why we preferred to use Ubuntu.

Lines 3 to 5 install Python3 and pip (which is used to install Python dependencies).

Line 7 is used to copy the requirements file from the local machine to the Docker image, and line 8 then installs all the dependencies listed in that file.

Line 10 then copies the rest of the files to the Docker image, and line 11 makes this directory the working directory. Finally, line 13 executes the script corresponding to the launcher agent described in 3.7.3 and passes the command line arguments.

Once the Dockerfile was created, we only had to use the `docker build` command in order to create the image.

The image was then pushed onto Docker Hub<sup>6</sup>, which is a hosted repository service used to store container images and be able to share them easily. Therefore, since the repository was made public, anyone can access this container, and execute the solution in a very easy way.

In order to execute the container, a user with Docker installed on his machine first just has to pull the repository :

```
docker pull miromatagne2103/tfm-spade-agents:latest
```

Then, in order to execute the application, the only command to execute is :

```
docker run -it --net=host miromatagne2103/tfm-spade-agents:latest  
--interface-port <port>
```

where `<port>` is the number of the port where the graphical interface of the launcher agent will be accessible.

This interface is always available in the user device's browser at the address `127.0.0.1:<port>/agent`

The `--net=host` argument indicates that Docker host networking is used, which means that Docker shares the same namespace as the host machine. This is useful in this situation because the graphical interfaces of the agents are available on certain specific ports, and need to be accessible on the host machine. Therefore, by using this option, the agent interfaces are available on the specified ports without any problem and without needing to specify port redirections to Docker. Indeed, the previous solution was to redirect ports between Docker and the host machine, but this meant that either we had to redirect a lot of ports (which takes a lot of time and memory), or the user could only choose from a selection of ports for the interfaces, which makes the application less user-friendly.

**IMPORTANT :** Note that the execution will only work if the user device is connected to a UPV network, or to a VPN to a UPV network. Indeed, the `gtirouter.dsic.upv.es`

---

<sup>6</sup><https://hub.docker.com/>

XMPP server is used in this application, which is only accessible from a UPV network. If the user is not connected to a UPV network, the application will not work.



---

---

## CHAPTER 4

# Experimental results

---

Once the Asynchronous Decentralized Federated Learning algorithm was defined and the application running this algorithm was implemented, experiments had to be made in order to first of all test if the solution works appropriately. Then, it is important to evaluate if this solution presents advantages compares to the other versions of Federated Learning. In these experiments, the MNIST database will be used.

### 4.1 Evaluation of the solution

---

First of all, the solution had to be tested in order to see if the agents communicate well between themselves, and if the training process is carried on appropriately.

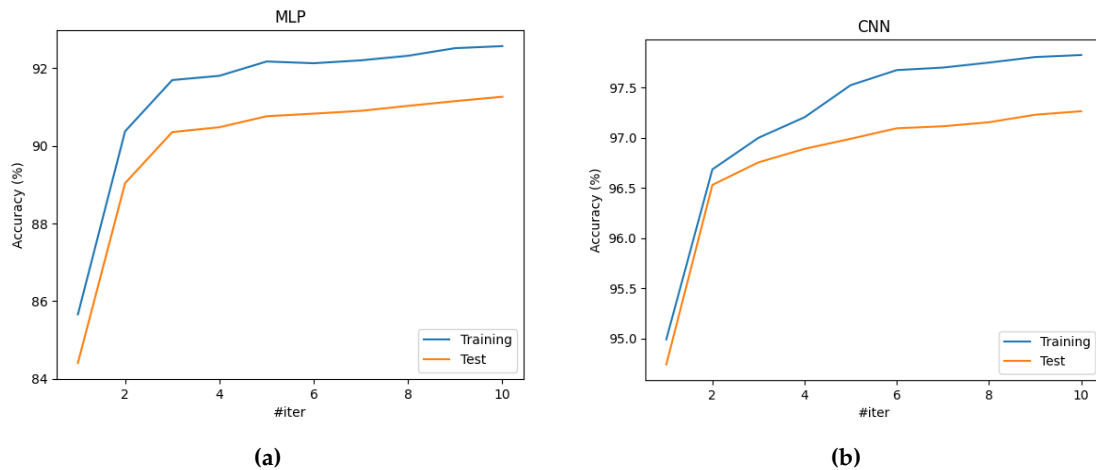
In order to verify this, we used the logs that were created in order to keep a trace of all sent messages, all training data after each round, all variations of the  $\epsilon$  parameter, as well as all the weights of the local model after each change (training or consensus).

The first tests were carried out on a single machine (a MacBook Pro with a 1.4 GHz Quad-Core Intel Core i5, 8GB of RAM), with 2 running agents that are both connected one to another in the network graph. In order to evaluate if the training was working as desired or not, 10 identical experiments were made, where each agent would run for 10 training rounds. The test accuracies were stored in the logs, and we could follow the evolution of the training process throughout the rounds by writing some scripts that create graphics using Matplotlib<sup>1</sup>, which is a Python library used to create visualizations. Note that in this experiment, the used MLP contains 32 neurons in the hidden layer (the  $x$  parameter in figure 3.4 is equal to 32). The results are plotted in figure 4.1.

For both models (MLP and CNN), the test accuracy is clearly increasing as the number of rounds increases, which tends to indicate that the training process is working properly. We also notice that in both cases the training accuracy is slightly higher than the test accuracy, which is a sign of a slight overfitting, which is very common during ML model trainings. The test accuracies reached after 10 rounds are around 91% for the MLP and around 97.2% for the CNN. By comparing these values to the various benchmarks present in [37], we realize that the obtained results seem coherent. Of course, we do not reach the maximal test accuracies ever registered, because training only 10 rounds is quite short

---

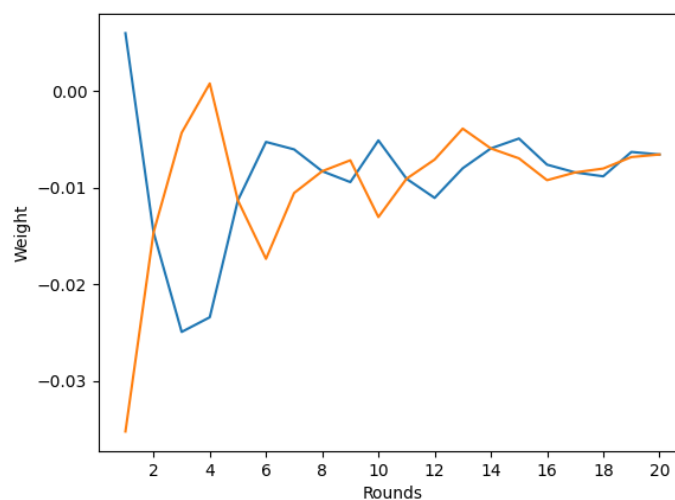
<sup>1</sup><https://matplotlib.org/>



**Figure 4.1:** Evolution of training and test accuracy in function of the number of iterations, both for a Multi-Layer Perceptron (a) and for a Convolutional Neural Network (b)

and because we did not try to tune the parameters of the model.

In order to verify that the consensus is reached, another test to carry is to check if the weights of the models of the 2 agents are converging towards the same values. In order to test this, we simply analyze the logs containing the weights of the models of the agents and plot the evolution of the values of one of the weights (the same one) for both agents. The result is presented in figure 4.2, and as we see, the weights of both agents converge throughout the training process, which indicates the consensus algorithm seems to work properly and a consensus seems to be reached after a certain time.



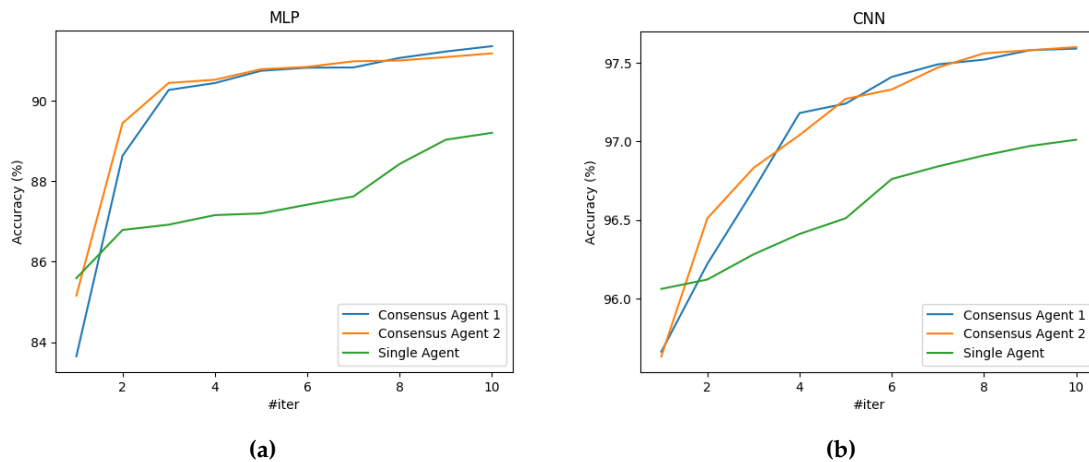
**Figure 4.2:** Evolution of one MLP weight of 2 agents throughout 20 rounds of training

## 4.2 Comparison with a single training agent

A comparison was made in order to compare the Asynchronous Decentralized Federated Learning setting with a single client training the model in local (without any communi-



cation with any other agent, and without handling any receiving messages). In the ADFL setting, 2 agents were used, they kept exchanging their weights throughout the process, and applied the consensus as described previously. This setting therefore has 2 times more computational power than a single agent training a model locally, but if the consensus algorithm was to be inefficient, a clear difference in performance could not be observed. The evolution of the test accuracy throughout 10 rounds of training for both situations was plotted and is represented at figure 4.3. Note that the displayed data is the result of 10 simulations, averaging the accuracies for each round in order to obtain analyzable data.



**Figure 4.3:** Evolution of training and test accuracy for 2 agents applying the ADFL algorithm and one single agent training locally in function of the number of iterations, both for a Multi-Layer Perceptron (a) and for a Convolutional Neural Network (b)

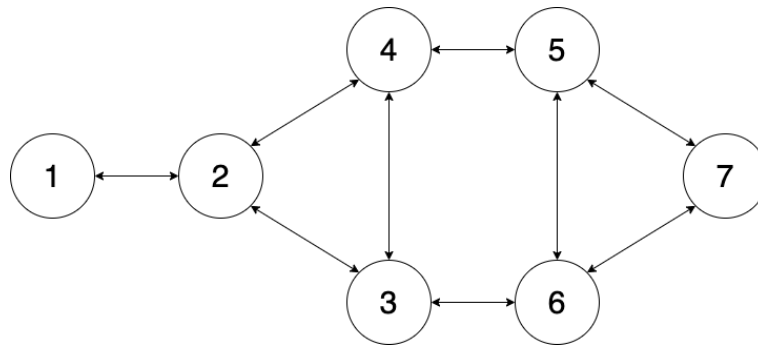
For both the MLP and the CNN, the 2 agents using the ADFL setting and applying the consensus algorithm present significantly higher test accuracies than the single agent training locally, which indicates that the consensus process seems to be efficient and allows multiple agents to combine their weights in a way that improves the global performance. Indeed, the single agent does not manage to achieve high test accuracies after 10 rounds, whereas the agents applying ADFL seem to stagnate around 91% for the MLP and around 97.5% for the CNN.

### 4.3 Comparison with Synchronous Decentralized Federated Learning

The main goal of the study is to determine if the Asynchronous Decentralized Federated Learning setting presents advantages compared to the Synchronous Decentralized Federated Learning setting. Indeed, the Synchronous Decentralized Federated Learning setting had already been created and tested before the development of the Asynchronous Decentralized Federated Learning setting, in the hopes that the latter would represent an improvement in performance due to the absence of waiting times, which is considerable in the synchronous setting.

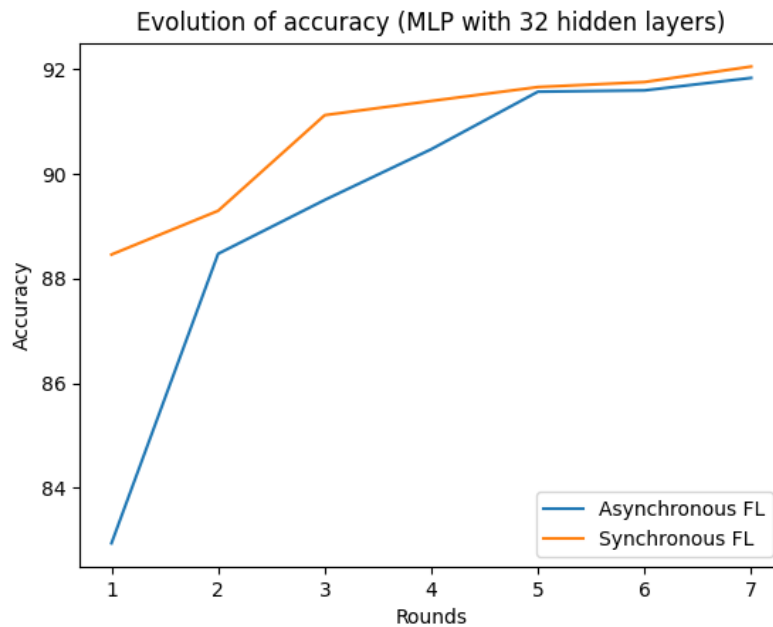
The first experiments were conducted with the graph presented in figure 4.4, which has a maximum degree of 3 (indeed, the nodes 2, 3, 4, 5 and 6 have 3 neighbours). Note

that these tests were carried out in the DSIC computer laboratories, with one agent running on each machine. The MLP model used throughout the experiments have 32 neurons in the hidden layer (the  $x$  parameter in figure 3.4).



**Figure 4.4:** Graph representing the network of agents used to conduct experiments to compare the ADFL setting to the SDFL setting

The 7 agents were launched both with the synchronous and asynchronous setting and the results of the trainings after each round were stored in logs. The results of the evolution of the test accuracy round by round is displayed in figure 4.5. Note that these accuracies represent the average of all test accuracies of the 7 agents round by round. Furthermore, the experiments were conducted 5 times, and averaged, in order to obtain more reliable results.

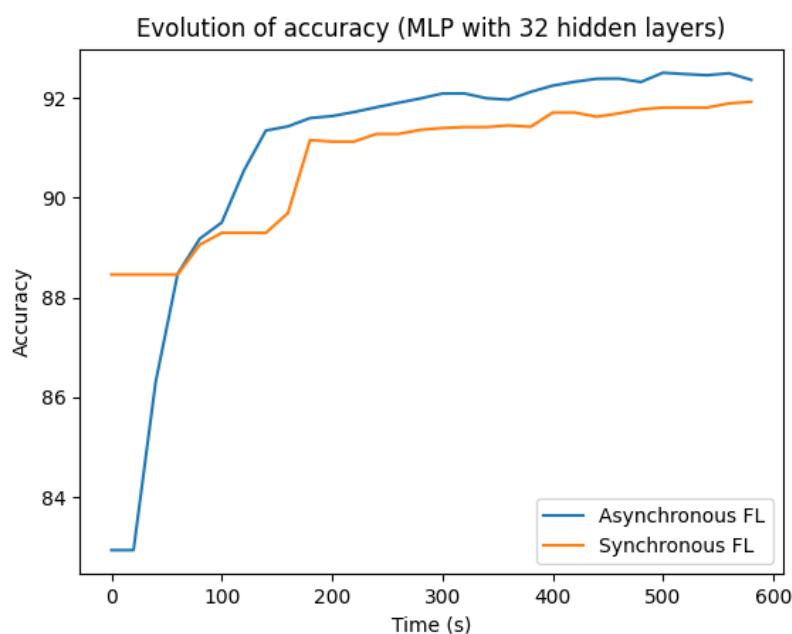


**Figure 4.5:** Evolution of test accuracy round by round for the synchronous and asynchronous setting

As we can see, the performance of the synchronous version seems better throughout the 5 first rounds, then is more or less equal to the performance of the asynchronous version. This is an expected result as at each round, the synchronous version takes into account the model of all of its neighbours, whereas the asynchronous version applies

consensus with only one agent, plus the agents that sent him weights during its training phase (no equal or lower than its number of neighbours). Therefore, the consensus performed in the synchronous version takes into account more training data, and gives place to a more efficient model in terms of test accuracy.

However, the advantage of Asynchronous Decentralized Federated Learning lies in the fact that it does not present long waiting times, because of its asynchronous character. It is therefore primordial to compare the performance of the synchronous and asynchronous versions in regard to execution times, in order to see if the gain of time of the asynchronous setting compensates the more efficient training of the synchronous version at each round. The evolution of test accuracy in regard to time for both asynchronous and synchronous versions are plotted on figure 4.6.



**Figure 4.6:** Evolution of test accuracy in function of time for the synchronous and asynchronous setting

As we can see, after 50 seconds, the asynchronous version presents a higher test accuracy than the synchronous version throughout the execution. The fact that the synchronous version seems more efficient in the first 50 seconds can be explained by the fact that the start of the timing (second 0) corresponds to the end of the first training round (because before that moment, no test accuracies have been obtained yet). Therefore, as a round of training is more efficient in the synchronous setting than in the asynchronous setting, it is normal that the synchronous version presents a higher value of accuracy at the beginning of the measurements.

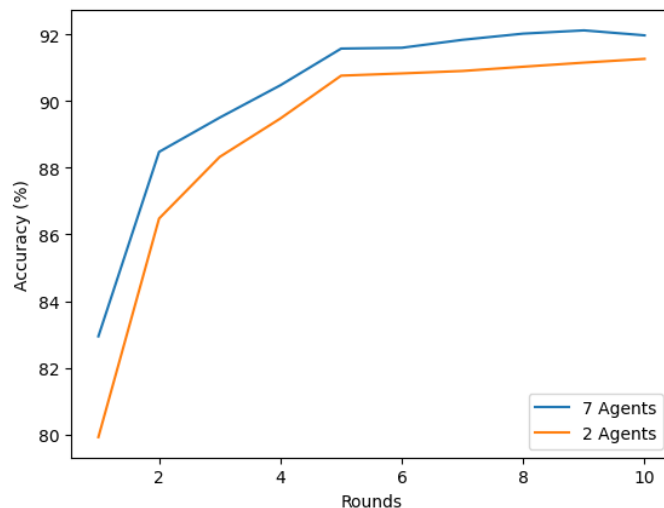
In conclusion, the Asynchronous Decentralized Federated Learning setting presents a lower efficiency than the synchronous version when comparing the performances round by round, but does present an advantage when comparing the accuracies in regard to time. From a practical point of view, the objective when training an AI model is more often to train it fast, and not limit the number of rounds, which makes of Asynchronous

Decentralized Federated Learning a very interesting and efficient solution.

It is however very important to stress that these results are specific to the agent network graph of figure 4.4. Indeed, it is not possible to generalize these results to any situation. Indeed, the performance of the SDFL setting highly depends on the topology of the network graph. If the maximum degree of the graph is higher, the time the agents will have to wait to obtain all the weights of their neighbours will be higher, and therefore the performance in regard to time is altered. Therefore, we could expect a better performance from the SDFL algorithm when the experiments are run with a graph with a maximal degree of 2 for instance. Furthermore, with a graph of degree higher than 3, we could expect that the difference in the performance of both algorithms would be even greater and that the Asynchronous Decentralized Federated Learning would be even more efficient in regard to the synchronous version.

#### 4.3.1. Influence of the number of agents

The number of agents in the network agent graph participating to the Federated Learning algorithm is also a factor that influences the performance of the solution. Indeed, the most intuitive answer is that the more agents there are, the more computing power is available, therefore the more performant the solution is. In order to verify this, an experiment was made comparing the performance of an execution with 2 agents to the performance of an execution with 7 agents. The model used for this experiment is a MLP with 32 neurons in the hidden layer and training on the MNIST dataset. The experiment was conducted 10 times and averaged, the test accuracy is plotted for both systems in figure 4.7.



**Figure 4.7:** Evolution of average test accuracy in function of the number of rounds for a system with 2 agents and a system with 7 agents

More experiments would have been made in this regard in order to have more complete results with systems with other numbers of agents, but the availability of the computer laboratories was limited during the project. In any case, the results in figure 4.7 show clearly that the system with 7 agents is more performant than the system with 2 agents. This makes sense as the computing power of the system with 7 agents is higher,

and the ADFL algorithm allows to quite quickly share the models between agents and to therefore obtain good quality model in a shorter time.



---

---

## CHAPTER 5

# Limitations and bottlenecks

---

Although the implemented solution works perfectly and meets the objectives of this work, it is important to analyze its limitations, which are common in Federated Learning implementations and remain open problems today. In order to understand the limitations and the causes of possible bottlenecks of the Decentralized Asynchronous Federated Learning setting, it is necessary to establish a list of factors that might influence the performance of the solution, and that could be the cause of bottlenecks.

### 5.1 Length of the messages

---

First of all, it is reasonable to suppose that the length of the messages to be sent influences the execution times of the solution, and therefore its efficiency. Since the transmitted messages contain weights and biases of the agents' models, the length of the messages is therefore dependant on the number of weights to be transmitted. In order to illustrate the evolution of the length of the messages in regard to the complexity of the model, measures were made with the MLP model previously presented (figure 3.4). In this model, the number of neurons in the hidden layer was called  $x$ . When the number of neuron increases ( $x$  increases), the model complexity increases as well. There are therefore more weights contained in the model and more weights to be transmitted, giving place to longer messages. The length of the messages to be transmitted depending on the  $x$  parameter are shown in table 5.1. It is important to note that, as stated previously, the transmitted messages contain the weights of the model but also information about the  $c$  parameter (the maximal degree of the graph is transmitted).

$x$	Length of the message
8	36 480
16	70 846
24	105 225
32	139 591
40	173 957
48	208 324
56	242 690
64	ERROR

**Table 5.1:** Length of the sent messages in function of the  $x$  parameter of the MLP

As we can see, when the number of neurons in the hidden layer of the MLP is of 64, an error appears and the message can not be sent. This is due to the fact that the model is so big that the large number of weights is too long to be sent in one message. There are solutions that could be imagined to solve this problem, such as dividing the message into 2 parts and sending 2 separate messages for instance, but this would of course induce more latency in the execution.

A first limitation of the solution is therefore that messages that are too large can not be sent from an agent to another.

It is also interesting to see if the length of the message influences the time the message takes to arrive until its destination. In order to analyze this, some tests were performed with messages of different lengths, measuring the time between the moment the message is sent and the moment the message is received by the receiving agent. The table 5.2 illustrates the results of this experiment that was run 50 times for each message length, showing the average time of message dispatching, as well as the maximal and minimal times recorded.

$x$	Avg dispatching time (s)	Min dispatching time (s)	Max dispatching time (s)
8	0.124	0.074	0.422
16	0.174	0.085	0.476
24	0.202	0.095	0.581
32	0.233	0.111	0.609
40	0.248	0.123	0.626
48	0.322	0.133	0.812
56	0.495	0.211	1.020

**Table 5.2:** Average, minimum and maximum times taken to send a message in function of the length of this message

As we can see, the time taken to send a message depends on the length of the message, as expected. A larger message takes a longer time to be sent, and furthermore the variance of the dispatching time seems to increase as well. Indeed, we can remark that the maximal dispatching time grows much faster than the average dispatching time, which indicates that when the message is longer, the time it takes to be sent is not only longer but also less consistent.

However, we can notice that the difference between the average dispatching time of a model with 8 neurons in the hidden layer and a model with 56 neurons in that hidden layer is only of 0.371s. This value could possibly be negligible in comparison with the time that the training of the model takes. In order to verify this, the training times of the models were measured over 50 iterations and averaged, the results are displayed in table 5.3.

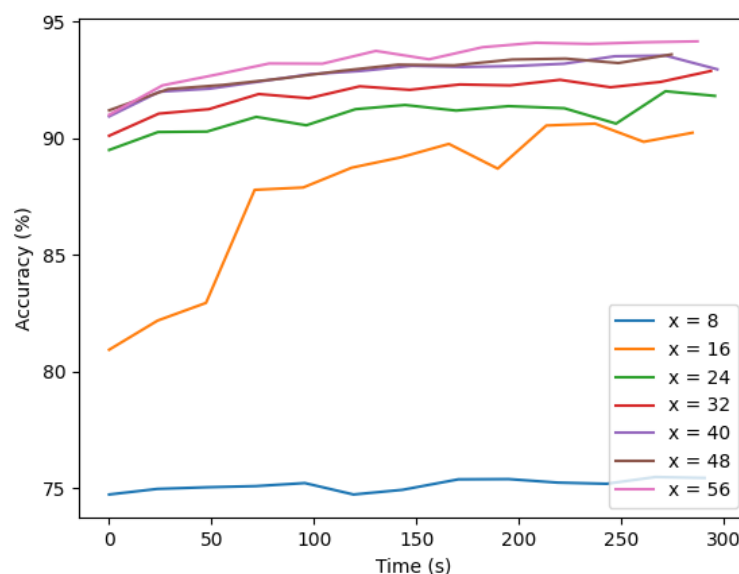


$x$	Avg training time (s)	Min training time (s)	Max training time (s)
8	23.741	23.505	23.908
16	23.829	23.619	24.311
24	24.548	24.309	25.025
32	24.620	24.291	25.271
40	25.154	24.932	25.513
48	25.649	25.400	25.897
56	27.260	25.808	29.715

**Table 5.3:** Duration of the MLP model training on the MNIST dataset in function of the size of the model

As we can see, the training times are considerably higher than the communication times presented in table 5.2. Therefore, in this specific case of training on a large dataset such as MNIST, the differences in the communication times are negligible compared to the training time of the models. The length of the messages can therefore not be considered as a limitation in this case since the difference in dispatching times is negligible.

Furthermore, table 5.3 puts in evidence that the training times increase as the size of the model increases. The difference between the smallest MLP model tested (with 8 neurons in the hidden layer) and the largest one (56 neurons in the hidden layer) is of 3.5 seconds, which could not be neglected in certain situations. It is therefore interesting to compare the results obtained with these different models in order to understand which value of  $x$  is the best in order to combine an efficient model in regard to execution time and an accurate model. An experiment was made analyzing the evolution of test accuracy during 250 seconds for 7 models with different numbers of neurons in the hidden layer of their MLP, the results are plotted in figure 5.1.



**Figure 5.1:** Evolution of test accuracy in function of the number of neurons in the hidden layer of the used MLP

As we can see, as the models contain more neurons in the hidden layer, the higher the test accuracies are. The model containing only 8 neurons in the hidden layer is unusable, the model is too simple and is not able to learn, which explains why the accuracy does not increase with time. We can also notice that even though the training times are higher as the model complexity increases (as explained in table 5.3) and the communication times increase too (as explained in table 5.2), the more complex models are still more efficient in regard to time : they present a higher test accuracy at all times even though they are slower models.

Therefore, this means that in this particular the developed solution does not limit the usage of more complex models, since the benefit in performance that they present outweighs the increase in communication time and training time. However, there is still a limitation in the fact that messages that are too long can not be sent, as shown in table 5.1 where the weights of a 64 neuron hidden layer MLP could not be sent.

## 5.2 Number of agents running on the same machine

Another factor that could influence the performance of the solution is the number of agents that are running on the same machine. Indeed, training a ML model requires computational resources, and when a lot of agents are training their models simultaneously on the same machine, they have to share these resources. Therefore, the training times may be slower when more agents run on the same machine. In order to verify this statement and evaluate how big of a difference in training times is observed when changing the number of agents running on a single machine, experiments were made with a 32 neuron hidden layer MLP. For different amounts of agents running simultaneously on the same machine, the average, maximal and minimal training times were measured and are displayed in table 5.4.

Number of agents	Avg training time (s)	Min training time (s)	Max training time (s)
1	24.620	24.291	25.271
2	26.477	26.178	31.416
3	30.720	29.514	32.345
4	34.993	33.649	35.695

**Table 5.4:** Duration of the MLP model training on the MNIST dataset in function of the number of agents training on the same device

As we can see, the differences in the training times when the number of agents running on the same device increases are significant. Indeed, there is nearly a 50% increase in the average training time when there are 4 agents training on a single device compared to the case where only a single agent is running.

Therefore, the number of agents running on a single device is clearly a factor that reduces the performance of this solution. In all experiments that were made, it was always made sure that when comparing results, the number of agents running on the same device in different experiments was the same. In practise, the ideal case is of course when

a different device is attributed to each agent, which was the case for the tests that were made in the computer labs.

### 5.3 XMPP server

---

As mentioned previously, the solution relies on the XMPP protocol for the inter-agent communication. This induces the presence of a XMPP server, to which the agents will send the messages, and that will be dispatched to the other agents later. Therefore, in the case where all the agents are on the same XMPP server, all the messages transit through this server, which may represent first of all a Single Point of Failure (SPoF) but also a bottleneck in terms of performance. Let us remember that the main advantage of the decentralized FL architecture is that there is no more central server, the presence of this central XMPP server that dispatches the messages can therefore be a little controversial. However, the role of a central server in a centralized FL architecture and the role of a XMPP server are very different. The XMPP server simply dispatches the messages, and keeps in memory the contacts of each agent. It does not have to perform model aggregation or other operations that require a lot of computing power. The operations performed by the XMPP server are very fast, and in contrary to a central server in a centralized architecture, the messages do not all arrive at the same time to the server, which reduces already a bit the risk of having a bottleneck.

However, we can not deny that the quality of the XMPP server is an important criteria to consider. It should have the following properties :

- Receive, process and dispatch the incoming messages quickly.
- Be able to receive a large number of messages at the same time.
- Process multiple messages in parallel



---

---

# CHAPTER 6

## Methodology

---

### 6.1 Tools

---

Multiple tools were used throughout this project, which were carefully chosen to optimize efficiency and performance.

#### 6.1.1. Notion

A Notion<sup>1</sup> workbook was used throughout the study, in order to organize the work done so far, as well as track the current progress and the tasks that remain to be done. This workbook contained several pages :

- A To-Do list that allows to track the tasks to do, the ones being treated at the moment and the completed ones.
- A Calendar that keeps track of all team meetings and deadlines, and contains summaries of all meetings, as well as the next working steps discussed in these meetings.
- A Redaction section, that gathers ideas throughout the work in order to organize and prepare the redaction of the final report.

Figure 6.1 shows an example of a notion page summarizing a team meeting.

#### 6.1.2. PyCharm

As the implementation is in the Python programming language, the IDE (Integrated Development Environment) used throughout this work is PyCharm<sup>2</sup>. This choice is justified by its ease to use and its integrated work environments that allow easy installations of packages.

Furthermore, Pycharm provides support for documenting, structuring and commenting the code following the PEP8<sup>3</sup> code conventions. Indeed, the whole project was made

---

<sup>1</sup><https://www.notion.so/>

<sup>2</sup>[www.jetbrains.com/pycharm/](https://www.jetbrains.com/pycharm/)

<sup>3</sup><https://peps.python.org/pep-0008/>

## Catch-Up 15

📅 Date September 27, 2022

🏷️ Tags Empty

+ Add a property

---

🗨️ Add a comment...

---

**Done since last meeting :**

- The consensus-based approach for the epsilon variable has been implemented and tested.
- Graphs with losses plotted instead of accuracies have been produced.
- CNN model has been tested and produced similar results to the MLP.
- Some scripts and logs have been made ready for the lab testing in September.

**To Do :**

- Add Nightingale chart to agent interface to show the number of sent/received messages per neighbour
- Add option to load model file from launcher agent
- Check if it is possible to kill the launcher agent after he created the training agent

📌 🚫 When trying to kill the launcher agent, an error of "maximum recursion depth exceeded" appears. It seems to be a problem with the web interface, since if we close the web interface while the agent is trying to stop the error appears immediately.

- Figure out how to execute the synchronous version of CoLearning

📌 🗣️ The Consensus-FDL branch of the Github repository is the one to execute for the tests. The `Connection_1.csv` file needs to be modified to fit the graph that we used for the asynchronous version. The files `Agent_1.py`, `Agent_2.py`, ... have to be executed, and then finally the file `Consensus_Learning_Initiator.py`, which will send an initial message to all the agents telling them who they should subscribe to.

**Figure 6.1:** Example of a team meeting summary on Notion

following these convention and allows other developers to have a quick and easy understanding of the code.

### 6.1.3. Git

Git<sup>4</sup> is a version control system that was used throughout this project to store the code and control the different versions of the solution. In particular, Github<sup>5</sup>, which is an Internet hosting for version control making use of Git was used. This allows the different contributors to the project to be able to see the advances made throughout the process.

### 6.1.4. Zotero

The research part of this project constituted a major part of the work, and had to be properly organized in order to not get confused with the amount of information to process, and to be able to write a clear report. In order to achieve this, Zotero<sup>6</sup> was used throughout the work. Zotero is free and open source reference management software, that allows

<sup>4</sup><https://git-scm.com/>

<sup>5</sup><https://github.com/>

<sup>6</sup><https://www.zotero.org/>

to very efficiently organize references used for a study. Most of the references are automatically added, simply by entering the ISBN, the DOI, the PMID, the arXiv ID, or the ADS bibcode of the reference. The relevant data about the reference is then automatically filled out.

These references can then be exported, to be used in a Microsoft Word or  $\text{\LaTeX}$  document for example.

### 6.1.5. Overleaf

Overleaf<sup>7</sup> is a free online  $\text{\LaTeX}$  editor. The report was written in  $\text{\LaTeX}$  for its easiness to write mathematical equations and for the flexibility it brings when writing a long report.

Overleaf also allows to automatically sync the Zotero references into the final document, and it also creates the Reference section of the report automatically.

## 6.2 Project Management

---

A meeting with my supervisors was organised every week at the beginning of the project, in order to track the progress of the work, and establish what were the next steps. At each meeting, I would explain the advances that were made since the last meeting, the problems I have encountered, ask the questions I may have about this work, and propose some next steps and objectives. As the project moved on, these meetings became less frequent as I got more comfortable working on this project, the guiding lines were more clear and the tasks to achieve were sometimes of a larger scope and required more time. All the content of these meetings was marked down in the Notion document.

At some points in the project, as explained in the report, some experiments had to be carried out and sometimes this could not be done on a single machine. Therefore, computer labs of the DSIC department at the UPV were booked in order to perform these experiments and deduce results about the developed solution.

The requirements of the project and risks involved were constantly kept track of, and updated when necessary. When new risks would appear, they would immediately be treated (finding ways to mitigate it).

---

<sup>7</sup><https://www.overleaf.com/>





---

---

## CHAPTER 7

# Further improvements

---

Although the developed solution is completely functional and can be used to perform Asynchronous Decentralized Federated Learning, there are still some further improvements that could be made to it, which were outside the scope of this work. Furthermore, concerning the themes of Federated Learning and in particular Asynchronous Decentralized Federated Learning, a lot of research is still made to this day in order to find reach better performing algorithms.

### 7.1 Improving the current solution

---

#### 7.1.1 Limiting the length of the messages

As seen in section 5.1, the length of the messages can be a factor that weakens the performance of the solution, although in the case that we studied it is negligible because the difference in time caused by the length of the message is very small compared to training times. However, we could think of settings where the dataset is much smaller, or where the models are way more simple, and where the training would be quicker, and in that case the length of messages could be an influential factor. Furthermore, let us not forget that, as explained in section 5.3, the XMPP server can be a bottleneck when transmitting messages, and if the messages are larger the risk of bottleneck could increase. Furthermore, as we saw in section 5.1, there is a maximal length above which a message can't be sent, which means some complex models are not usable with the current solution. Therefore, in any case, trying to limit the length of messages could be useful.

One way to reduce the length of the messages could be to use rounding in the weights of the model. Indeed, the weights stored in the model are decimal numbers, with a precision of 18 decimals. This occupies a lot of memory, and we could think of rounding these weights to less decimals. Of course, this makes the model less precise and we could expect a decrease in the performance. The increase in performance obtained by reducing the execution times has to be compared to the decrease of performance due to the loss of precision in the model weights, in order to conclude if this method improves the solution or not.

Another solution to limit the length of the messages is to use matrix compression. Indeed, as explained in [54], there are several matrix compression techniques that exist, and that can efficiently reduce the size of a matrix. For example, a popular compression method is pruning, where all the weights that are very close to 0 are replaced by a 0,

which considerably reduces the size of the matrix.

### 7.1.2. Launch all agents at once

Another drawback of this implementation is that all agents have to be executed individually. This, of course, seems logical from a resources point of view. Indeed, the agents consume a lot of resources and are most efficient when ran on separate machines (see section 5.2) so it makes sense to execute each agent by itself. However, when the agent network graph is very large and the number of agents is large, launching all the agents could prove to be a long process. Therefore, some kind of automation of this process could be implemented, even though it would imply several devices. We could imagine that the user specifies in an interface which devices he wants to use, as well as the agent network graph, and that the code would automatically connect with these devices and launch each agent automatically on all the devices.

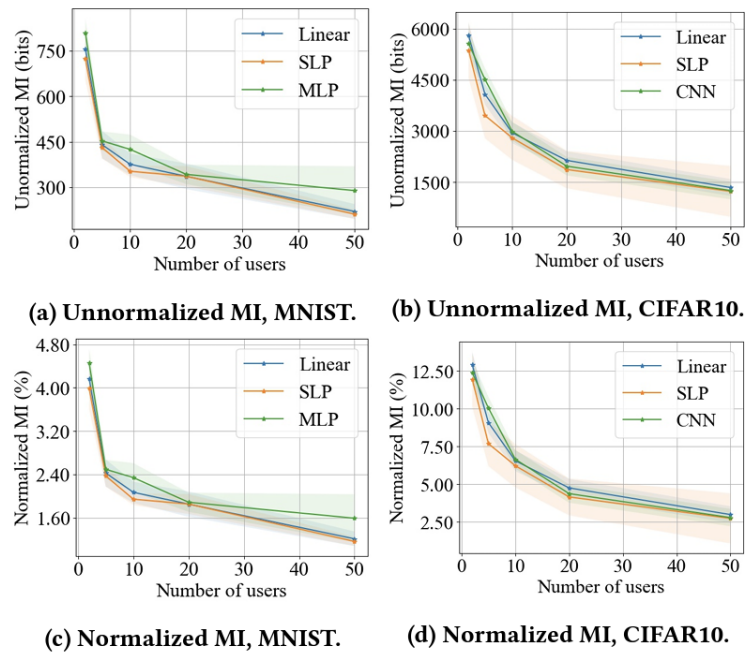
## 7.2 Future works

---

A lot of research nowadays is conducted on an aspect of Decentralized Federated Learning that was not focused on in this work, which is privacy aspect [8]. Indeed, we could imagine a setting where each of the agents possess private data that can not be shared with any other agent. During the Federated Learning process, the agents communicate weights, which are not private pieces of information. However, after an agent performs model aggregation, some information about the other agent's dataset could be figured out. Indeed, Elkody et al. [18] showed that the information leakage is sometimes non negligible, and that it is higher when the amount of participating agents is lower. Their investigation were conducted on a Centralized Federated Learning setting, which means that model aggregation was performed on a central server with all the weights of all the agents. Therefore, it makes sense that if the number of agents is high, there are contributions from so many agents in the final model that it is difficult to identify the contribution of a single agent's model, and that makes the privacy leaks lower. On the other hand, when the number of participating agents is low, it is easier to identify the contributions of each agent to the final model. Figure 7.1 shows the results obtained in [18] when making experiments measuring the evolution of Mutual Information in function of the number of clients present in the Federated Learning setting. Mutual Information (MI) effectively represents the privacy leakage, as it measures the mutual information between an agent's model and the output model after the model aggregation phase. The tests were made with 3 models : a SLP (Single-Layer Perceptron, which is just like a MLP but with one layer), a MLP and a CNN. As we can see, the Mutual Information decreases as the number of clients increase.

However, in the Asynchronous Decentralized Federated Learning Architecture presented in this study, the model aggregation is always performed on only 2 agents' weights at a time, as explained previously. Therefore, following the conclusions of Elkody et al. [18], this could lead to a large amount of privacy leakage. This is therefore an aspect of the elaborated solution that should be investigated in order to remove privacy concerns.

Another aspect of Federated Learning that is studied is the defense against attacks. Indeed, a number of attacks such as data poisoning [36], model update poisoning [4, 7]



**Figure 7.1:** Evolution of normalized and unnormalized Mutual Information in function of the number of users in the Federated Learning setting, with 3 different models. This figure was taken from [18].

and model evasion attacks exist [21, 36]. Without going into details into these different attacks, we can illustrate a very simple one. If a malicious agent sends weights to a running agent and no security is implemented, the agent will perform model aggregation with erroneous weights and give bad results. We could even think that the malicious agent would send weights that influence the outcome of the model in a malicious way. There are already some defenses against these attacks as explained in [31], but the decentralized architecture makes it more difficult to implement. Therefore, some research could be made on this subject to make the implemented solution more secure against attacks.



---

---

## CHAPTER 8

# Conclusion

---

### 8.1 Synthesis

---

The numerous objectives of this work have all been met and the final obtained results are very satisfactory. Indeed, the first objective was to develop a Synchronous Decentralized Federated Learning implementation which was already started by other researchers before the start of this work. This was successfully achieved and proved to be very useful in order to evaluate the performance of the other implementation that was realized.

The second objective was to create a new Federated Learning algorithm that gathers the advantages of Decentralized Federated Learning and of Asynchronous Federated Learning, which we named Asynchronous Decentralized Federated Learning. This was a success since the algorithm was created, and then tested theoretically providing satisfactory results.

Next, the focus was on the creation of a standalone user-friendly application that allows to easily execute the Asynchronous Decentralized Federated Learning algorithm for an arbitrary configuration of agents. This represented the largest part of the work, and combined a lot of different themes to obtain the final product. This final application was evaluated, tested, and compared to the synchronous implementation that was created. As expected, the ADFL algorithm presented very good results, outperforming the synchronous algorithm in the tested cases. Furthermore, an emphasis was put on the usability of the application, which is why graphical interfaces monitoring the execution were implemented, along with an interface where the user could specify the parameters of the execution. In addition, the solution was containerized so that it could be accessed and executed very easily by any user. The application is therefore first of all performant, but also accessible and easy to use.

Finally, since a part of this project is dedicated to research and investigation, it is important to think about the impact of the produced work and of the possible future lines of work it enables. In this regard, this work enables other researchers to further pursue the work, in particular thanks to the publication of an academic paper sent to the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS) [12]. We believe the developed solution opens new doors in the domain of Federated Learning and could possibly be improved.

## 8.2 Relation with the pursued studies

---

This thesis is the perfect conclusion to the Master studies I am carrying out at ULB (Université Libre de Bruxelles) and UPV (Universitat Politècnica de Valencia) as it involves many concepts that were at the heart of my education.

First of all, the heart of the project focuses on Artificial Intelligence, which is the specialty my Master at ULB is focused on. The thesis was therefore a great opportunity to put into practise the theoretical knowledge I had learned about Artificial Intelligence in general, but also Machine Learning in specific. This allowed me to discover the state of the art algorithms in Federated Learning, which was a concept I was not familiar with before the start of the work, and I believe this to be a great discovery for my future professional career as it is one of the most promising technologies in this field.

Next, this thesis also allowed me to put into practise the skills I have learned throughout my time at UPV. Indeed, Multi-Agent Systems and in particular the SPADE Python library had been seen in class, but not to the extent that it was used in this thesis. This work has made me really use the knowledge I learnt through these classes, and pushed me to research further about these subjects. This project also involved web-development skills, which were required to create the graphical interfaces of the agents using HTML, CSS and Javascript. Furthermore, the organization of the work was also something that the Master classes really helped me with. Indeed, the course "*Planificación y Dirección de Proyectos*" in particular was very helpful in order to properly organize the work and to focus on the final expectations of the developed application. Finally, on the more practical side, the several subjects of the Master presenting the use of Docker helped to produce a final solution that meets the requirement of it being a user-friendly and standalone application.

# Bibliography

---

- [1] Rusul Abduljabbar, Hussein Dia, Sohani Liyanage, and Saeed Asadi Bagloee. Applications of Artificial Intelligence in Transport: An Overview. *Sustainability*, 11(1):189, January 2019.
- [2] Abien Fred Agarap. Deep Learning using Rectified Linear Units (ReLU), February 2019. arXiv:1803.08375 [cs, stat].
- [3] Asian Development Bank, Hubert Jenny, Yihong Wang, Asian Development Bank, Eduardo Garcia Alonso, Asian Development Bank, Roberto Minguez, and Asian Development Bank. Using Artificial Intelligence for Smart Water Management Systems. Technical report, Asian Development Bank, July 2020.
- [4] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. How To Backdoor Federated Learning, August 2019. arXiv:1807.00459 [cs].
- [5] Kanadpriya Basu, Ritwik Sinha, Aihui Ong, and Treena Basu. Artificial intelligence: How is it changing medical sciences and its future? *Indian Journal of Dermatology*, 65(5):365, 2020.
- [6] Floris Bex and Henry Prakken. Can Predictive Justice Improve the Predictability and Consistency of Judicial Decision-Making? In Erich Schweighofer, editor, *Frontiers in Artificial Intelligence and Applications*. IOS Press, December 2021.
- [7] Arjun Nitin Bhagoji, Supriyo Chakraborty, Prateek Mittal, and Seraphin Calo. Analyzing Federated Learning through an Adversarial Lens, November 2019. arXiv:1811.12470 [cs, stat].
- [8] Subrato Bharati, M. Rubaiyat Hossain Mondal, Prajoy Podder, and V. B. Surya Prasath. Federated learning: Applications, challenges and future directions. *International Journal of Hybrid Intelligent Systems*, 18(1-2):19–35, May 2022. arXiv:2205.09513 [cs, eess, math].
- [9] Bruce Buchanan. A (Very) Brief History of Artificial Intelligence. *AI Magazine*, December 2005.
- [10] Nanette Byrnes. As Goldman Embraces Automation, Even the Masters of the Universe Are Threatened. *MIT Technology Review*, July 2017.
- [11] Mathilde Caron, Piotr Bojanowski, Armand Joulin, and Matthijs Douze. Deep Clustering for Unsupervised Learning of Visual Features, March 2019. arXiv:1807.05520 [cs].
- [12] Carlos Carrascosa, Miguel Rebollo, Aaron Pico, Jaime A. Rincon, and Miro-Manuel Matagne. Asynchronous Consensus for Multi-Agent Systems and its Application to

- Federated Learning. *22nd International Conference on Autonomous Agents and Multiagent Systems*, October 2022.
- [13] Lijia Chen, Pingping Chen, and Zhijian Lin. Artificial Intelligence in Education: A Review. *IEEE Access*, 8:75264–75278, 2020.
- [14] Yujing Chen, Yue Ning, Martin Slawski, and Huzefa Rangwala. Asynchronous Online Federated Learning for Edge Devices with Non-IID Data, October 2020. arXiv:1911.02134 [cs].
- [15] Zhikun Chen, Jiaqi Pan, and Sihai Zhang. Asynchronous Federated Learning in Decentralized Topology Based on Dynamic Average Consensus. In *ICC 2022 - IEEE International Conference on Communications*, pages 2822–2827, Seoul, Korea, Republic of, May 2022. IEEE.
- [16] Ali Dorri, Salil S. Kanhere, and Raja Jurdak. Multi-Agent Systems: A Survey. *IEEE Access*, 6:28573–28593, 2018.
- [17] Chris Duckett. Zoom patches XMPP vulnerability chain that could lead to remote code execution. *ZdNET*, May 2022.
- [18] Ahmed Roushdy Elkordy, Jiang Zhang, Yahya H. Ezzeldin, Konstantinos Psounis, and Salman Avestimehr. How Much Privacy Does Federated Learning with Secure Aggregation Guarantee?, August 2022. arXiv:2208.02304 [cs, math].
- [19] European Commission for the Efficiency of Justice (CEPEJ). Justice of the future : predictive justice and artificial intelligence, September 2018.
- [20] Abigail Goldsteen, Gilad Ezov, Ron Shmelkin, Micha Moffie, and Ariel Farkash. Data Minimization for GDPR Compliance in Machine Learning Models. *AI and Ethics*, 2(3):477–491, August 2022. arXiv:2008.04113 [cs].
- [21] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples, March 2015. arXiv:1412.6572 [cs, stat].
- [22] Morten Goodwin, Kim Tallaksen Halvorsen, Lei Jiao, Kristian Muri Knausgård, Angela Helen Martin, Marta Moyano, Rebekah A Oomen, Jeppe Have Rasmussen, Tonje Knutsen Sjørdalen, and Susanna Huneide Thorbjørnsen. Unlocking the potential of deep learning for marine ecology: overview, applications, and outlook. *ICES Journal of Marine Science*, 79(2):319–336, March 2022.
- [23] Zainab H., Hesham A., and Mahmoud M. Internet of Things (IoT): Definitions, Challenges and Recent Research Directions. *International Journal of Computer Applications*, 128(1):37–47, October 2015.
- [24] Chaoyang He, Conghui Tan, Hanlin Tang, Shuang Qiu, and Ji Liu. Central Server Free Federated Learning over Single-sided Trust Social Networks, August 2020. arXiv:1910.04956 [cs, stat].
- [25] Lie He, An Bian, and Martin Jaggi. COLA: Decentralized Linear Learning, June 2019. arXiv:1808.04883 [cs, stat].
- [26] Trevor Hill. How Artificial Intelligence is Reshaping the Water Sector. May 2018.
- [27] Timothy O. Hodson, Thomas M. Over, and Sydney S. Foks. Mean Squared Error, Deconstructed. *Journal of Advances in Modeling Earth Systems*, 13(12), December 2021.



- [28] Michael N. Huhns, editor. *Distributed artificial intelligence. 1: ed. by Michael N. Huhns*. Pitman, London, repr edition, 1988.
- [29] Ida Arlene Joiner. *Emerging Library Technologies*. Elsevier, 2018.
- [30] Stanimir Kabaivanov and Veneta Markovska. Artificial intelligence in real estate market analysis. page 030001, Tomsk, Russia, 2021.
- [31] Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, Rafael G. L. D’Oliveira, Hubert Eichner, Salim El Rouayheb, David Evans, Josh Gardner, Zachary Garrett, Adrià Gascón, Badih Ghazi, Phillip B. Gibbons, Marco Gruteser, Zaid Harchaoui, Chaoyang He, Lie He, Zhouyuan Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, Tara Javidi, Gauri Joshi, Mikhail Khodak, Jakub Konečný, Aleksandra Korolova, Farinaz Koushanfar, Sanmi Koyejo, Tancrède Lepoint, Yang Liu, Prateek Mittal, Mehryar Mohri, Richard Nock, Ayfer Özgür, Rasmus Pagh, Mariana Raykova, Hang Qi, Daniel Ramage, Ramesh Raskar, Dawn Song, Weikang Song, Sebastian U. Stich, Ziteng Sun, Ananda Theertha Suresh, Florian Tramèr, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. Advances and Open Problems in Federated Learning. *arXiv:1912.04977 [cs, stat]*, March 2021. arXiv: 1912.04977.
- [32] Navleen Kaur, Supriya Lamba Sahdev, Monika Sharma, and Laraibe Siddiqui. BANKING 4.0: “THE INFLUENCE OF ARTIFICIAL INTELLIGENCE ON THE BANKING INDUSTRY & HOW AI IS CHANGING THE FACE OF MODERN DAY BANKS”. *INTERNATIONAL JOURNAL OF MANAGEMENT*, 11(6), June 2020.
- [33] Brendan S. Kelly, Conor Judge, Stephanie M. Bollard, Simon M. Clifford, Gerard M. Healy, Awsam Aziz, Prateek Mathur, Shah Islam, Kristen W. Yeom, Aonghus Lawlor, and Ronan P. Killeen. Radiology artificial intelligence: a systematic review and evaluation of methods (RAISE). *European Radiology*, 32(11):7998–8007, April 2022.
- [34] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, January 2017. arXiv:1412.6980 [cs].
- [35] Guanghui Lan, Soomin Lee, and Yi Zhou. Communication-Efficient Algorithms for Decentralized and Stochastic Optimization, February 2017. arXiv:1701.03961 [cs, math].
- [36] John Langford and International Machine Learning Society, editors. *Proceedings of the Twenty-Ninth International Conference on Machine Learning: held from June 26 to July 1 in Edinburgh, Scotland ... in conjunction with the 25th Conference on Learning Theory (COLT 2012)*. Madison, Wis, 2012.
- [37] Yann LeCun, Corinna Cortes, and Christopher Burges. The MNIST database of handwritten digits, 1998.
- [38] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. Pipe-SGD: A Decentralized Pipelined SGD Framework for Distributed Deep Net Training, January 2019. arXiv:1811.03619 [cs, stat].
- [39] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent, September 2017. arXiv:1705.09056 [cs, math, stat].

- [40] Zachary Chase Lipton, Charles Elkan, and Balakrishnan Narayanaswamy. Thresholding Classifiers to Maximize F1 Score, May 2014. arXiv:1402.1892 [cs, stat].
- [41] Qi Liu, Bo Yang, Zhaojian Wang, Dafeng Zhu, Xinyi Wang, Kai Ma, and Xinping Guan. Asynchronous Decentralized Federated Learning for Collaborative Fault Diagnosis of PV Stations. *IEEE Transactions on Network Science and Engineering*, 9(3):1680–1696, May 2022. arXiv:2202.13606 [cs].
- [42] Hossin M and Sulaiman M.N. A Review on Evaluation Metrics for Data Classification Evaluations. *International Journal of Data Mining & Knowledge Management Process*, 5(2):01–11, March 2015.
- [43] Spyros Makridakis. Accuracy measures: theoretical and practical concerns. *International Journal of Forecasting*, 9(4):527–529, December 1993.
- [44] Sébastien Marcel and Yann Rodriguez. Torchvision the machine-vision package of torch. In *Proceedings of the international conference on Multimedia - MM '10*, page 1485, Firenze, Italy, 2010. ACM Press.
- [45] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data, February 2017. Number: arXiv:1602.05629 arXiv:1602.05629 [cs].
- [46] H. Brendan McMahan and Daniel Ramage. Federated Learning: Collaborative Machine Learning without Centralized Training Data, June 2017.
- [47] Ramesh Medar, Vijay S. Rajpurohit, and B. Rashmi. Impact of Training and Testing Data Splits on Accuracy of Time Series Forecasting in Machine Learning. In *2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA)*, pages 1–6, PUNE, India, August 2017. IEEE.
- [48] Elon Musk and Stephen Hawking. Research Priorities for Robust and Beneficial Artificial Intelligence: An Open Letter. January 2015.
- [49] Angelia Nedic and Alex Olshevsky. Distributed optimization over time-varying directed graphs, March 2014. arXiv:1303.2289 [cs, math].
- [50] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation Functions: Comparison of trends in Practice and Research for Deep Learning, November 2018. arXiv:1811.03378 [cs].
- [51] Reza Olfati-Saber, J. Alex Fax, and Richard M. Murray. Consensus and Cooperation in Networked Multi-Agent Systems. *Proceedings of the IEEE*, 95(1):215–233, January 2007.
- [52] Keiron O’Shea and Ryan Nash. An Introduction to Convolutional Neural Networks, December 2015. arXiv:1511.08458 [cs].
- [53] Javier Palanca, Andres Terrasa, Vicente Julian, and Carlos Carrascosa. SPADE 3: Supporting the New Generation of Multi-Agent Systems. *IEEE Access*, 8:182537–182549, 2020.
- [54] Rina Panigrahy. Matrix Compression Operator, February 2022.
- [55] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani,

- Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library, December 2019. arXiv:1912.01703 [cs, stat].
- [56] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for Activation Functions, October 2017. arXiv:1710.05941 [cs].
- [57] Miguel Rebollo. *Análisis de redes de comercio mediante procesos de consenso (Tesis de máster)*. PhD thesis, Universidad Politécnica de Madrid, Madrid, 2013.
- [58] Sebastian Ruder. An overview of gradient descent optimization algorithms, June 2017. arXiv:1609.04747 [cs].
- [59] R.O. Saber and R.M. Murray. Consensus protocols for networks of dynamic agents. In *Proceedings of the 2003 American Control Conference, 2003.*, volume 2, pages 951–956, Denver, Colorado, USA, 2003. IEEE.
- [60] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. Technical Report RFC6120, RFC Editor, March 2011.
- [61] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 44(1.2):206–226, January 2000.
- [62] R. Sathya and Annamma Abraham. Comparison of Supervised and Unsupervised Learning Algorithms for Pattern Classification. *International Journal of Advanced Research in Artificial Intelligence*, 2(2), 2013.
- [63] Munindar P. Singh and Amit K. Chopra. The Internet of Things and Multiagent Systems: Decentralized Intelligence in Distributed Computing. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1738–1747, Atlanta, GA, USA, June 2017. IEEE.
- [64] Benjamin Sirb and Xiaojing Ye. Consensus optimization with delayed and stochastic gradients on decentralized networks. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 76–85, Washington DC, USA, December 2016. IEEE.
- [65] Student, Department of Information Technology, Engineering Faculty, Udayana University, Bali, Indonesia, Kadek Darmaastawan, I Made Sukarsa, and Putu Wira Buana. LINE Messenger as a Transport Layer to Distribute Messages to Partner Instant Messaging. *International Journal of Modern Education and Computer Science*, 11(3):1–9, March 2019.
- [66] Latanya Sweeney. Simple Demographics Often Identify People Uniquely. *Carnegie Mellon University, Data Privacy Working Paper 3.*, 2000.
- [67] M.V. Valueva, N.N. Nagornov, P.A. Lyakhov, G.V. Valuev, and N.I. Chervyakov. Application of the residue number system to reduce hardware costs of the convolutional neural network implementation. *Mathematics and Computers in Simulation*, 177:232–243, November 2020.
- [68] Paul Vanhaesebrouck, Aurélien Bellet, and Marc Tommasi. Decentralized Collaborative Learning of Personalized Models over Networks, February 2017. arXiv:1610.05202 [cs, stat].
- [69] Rajendra Verma. How WhatsApp works. *Medium*, September 2019.
- [70] Lloyd Watkin and David Koelle. *Practical XXPP*. Packt Publishing, Limited, Birmingham, September 2016. OCLC: 982652038.

- 
- [71] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms, September 2017. arXiv:1708.07747 [cs, stat].
- [72] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do, and Kaori Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*, 9(4):611–629, August 2018.
- [73] Dayong Ye, Minjie Zhang, and Athanasios V. Vasilakos. A Survey of Self-Organization Mechanisms in Multiagent Systems. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 47(3):441–461, March 2017.
- [74] Xue Ying. An Overview of Overfitting and its Solutions. *Journal of Physics: Conference Series*, 1168:022022, February 2019.
- [75] Zhilu Zhang and Mert R. Sabuncu. Generalized Cross Entropy Loss for Training Deep Neural Networks with Noisy Labels, November 2018. arXiv:1805.07836 [cs, stat].
- [76] Minghui Zhu and Sonia Martínez. Discrete-time dynamic average consensus. *Automatica*, 46(2):322–329, February 2010.

---

---

# APPENDIX A

## Log files

---

### A.1 Message log file

---

The message log files contain the following content everytime a message is sent or received :

- The time when the message was sent or received
- A word indicating if the message was sent or received, and if it is a response to a previous message or not
- The ID of the message
- The JID of the communicating agent (sender or receiver)

An example of a message log file is presented :

```
time,send_or_rcv,id,communicating_agent
2022-11-20 11:30:47.517603,SEND,55b53a0f-1ff3-453f-8b4a-a6dd5c3fe9bf,
  my_agent_2@gtirouter.dsic.upv.es
2022-11-20 11:30:47.906637,RECEIVE,99c456e8-e750-4bb5-98b3-8d5928e3f890,
  my_agent_2@gtirouter.dsic.upv.es/edxeykXa
2022-11-20 11:30:47.916086,SEND_RESPONSE,99c456e8-e750-4bb5-98b3-8d5928e3f890,
  my_agent_2@gtirouter.dsic.upv.es/edxeykXa
2022-11-20 11:30:48.069169,RECEIVE_RESPONSE,55b53a0f-1ff3-453f-8b4a-a6dd5c3fe9bf,
  my_agent_2@gtirouter.dsic.upv.es/edxeykXa
2022-11-20 11:31:13.926643,SEND,304dd840-7b24-4a52-908c-947c79aa3900,
  my_agent_2@gtirouter.dsic.upv.es
2022-11-20 11:31:14.101474,RECEIVE,7fff0d85-f14a-4290-90bd-d4181914e4e0,
  my_agent_2@gtirouter.dsic.upv.es/edxeykXa
2022-11-20 11:31:14.112035,SEND_RESPONSE,7fff0d85-f14a-4290-90bd-d4181914e4e0,
  my_agent_2@gtirouter.dsic.upv.es/edxeykXa
2022-11-20 11:31:14.299558,RECEIVE_RESPONSE,304dd840-7b24-4a52-908c-947c79aa3900,
  my_agent_2@gtirouter.dsic.upv.es/edxeykXa
2022-11-20 11:31:40.411015,SEND,043c64bd-0c37-495c-b339-f2ba79bc8138,
```

**Figure A.1:** Example of an message log file

## A.2 Training log file

---

The training log files contain the following content for after each training round :

- The time at which the training round ended
- The training accuracy (in %)
- The training loss
- The test accuracy (in %)
- The test loss

An example of a training log file is presented :

```
time,training_accuracy,training_loss,test_accuracy,test_loss
2022-11-13 21:39:34.461417,90.47,-0.9,89.5,-0.8873
2022-11-13 21:39:58.355688,91.78,-0.91,90.27,-0.898
2022-11-13 21:40:22.238674,91.9,-0.91,90.29,-0.8998
2022-11-13 21:40:46.242788,92.23,-0.92,90.92,-0.907
2022-11-13 21:41:10.683955,92.55,-0.92,90.56,-0.9042
2022-11-13 21:41:34.681417,92.55,-0.92,91.25,-0.9095
2022-11-13 21:41:58.843816,92.4,-0.92,91.43,-0.913
2022-11-13 21:42:23.779355,92.48,-0.92,91.19,-0.91
2022-11-13 21:42:49.345270,92.75,-0.93,91.38,-0.9123
2022-11-13 21:43:16.606762,92.92,-0.93,91.29,-0.9124
2022-11-13 21:43:41.824005,92.18,-0.92,90.63,-0.9046
2022-11-13 21:44:05.943045,93.35,-0.93,92.02,-0.9191
2022-11-13 21:44:30.078670,93.17,-0.93,91.82,-0.9182
2022-11-13 21:44:54.252646,93.15,-0.93,91.84,-0.9184
2022-11-13 21:45:18.571565,92.6,-0.92,91.29,-0.9118
2022-11-13 21:45:43.045143,92.92,-0.93,91.31,-0.9124
2022-11-13 21:46:08.881585,93.17,-0.93,91.77,-0.9173
2022-11-13 21:46:33.542121,92.8,-0.93,91.69,-0.9162
2022-11-13 21:46:57.687568,93.22,-0.93,91.9,-0.9193
2022-11-13 21:47:21.894854,93.47,-0.93,92.22,-0.9221
```

**Figure A.2:** Example of a training log file

## A.3 Weight log file

---

The weight log files contain the following content for after each training round and after each time the consensus algorithm is applied, in the case of a MLP model :

- The time at which the training round ended or the consensus was applied
- The weight of the first neuron of the first layer
- The bias of the first neuron of the first layer
- The weight of the first neuron of the second layer
- The bias of the first neuron of the second layer

An example of a weight log file is presented (note that the weights have been rounded to 3 decimals for displaying purposes) :

```
time,train_or_consensus,first_layer_weight,first_layer_bias,second_layer_weight,
second_layer_bias
2022-11-20 11:30:47.515560,TRAINING,-0.012,-0.024,0.683,-0.081
2022-11-20 11:30:48.074827,CONSENSUS,-0.024,-0.031,0.02,-0.094
2022-11-20 11:31:13.925984,TRAINING,-0.021,-0.041,-0.107,-0.117
2022-11-20 11:31:14.302717,CONSENSUS,-0.021,-0.02,0.412,-0.017
2022-11-20 11:31:40.409484,TRAINING,-0.032,0.005,0.568,0.03
2022-11-20 11:31:40.723950,CONSENSUS,-0.021,-0.029,0.124,-0.065
2022-11-20 11:32:06.763837,TRAINING,-0.016,-0.043,0.081,-0.054
2022-11-20 11:32:07.285924,CONSENSUS,-0.03,-0.003,0.498,-0.01
2022-11-20 11:32:33.167315,TRAINING,-0.044,0.028,0.626,-0.008
2022-11-20 11:32:33.587914,CONSENSUS,-0.025,-0.018,0.114,-0.067
2022-11-20 11:32:59.507949,TRAINING,-0.021,-0.027,-0.107,-0.085
2022-11-20 11:32:59.905522,CONSENSUS,-0.032,-0.0,0.51,-0.002
2022-11-20 11:33:25.884820,TRAINING,-0.033,0.002,0.814,0.046
2022-11-20 11:33:26.196448,CONSENSUS,-0.028,-0.01,0.154,-0.023
2022-11-20 11:33:52.134188,TRAINING,-0.025,-0.018,-0.02,-0.006
2022-11-20 11:33:52.474179,CONSENSUS,-0.031,-0.004,0.624,0.038
2022-11-20 11:34:18.457201,TRAINING,-0.036,0.008,0.841,0.071
2022-11-20 11:34:18.817635,CONSENSUS,-0.023,-0.022,0.212,0.059
2022-11-20 11:34:44.761239,TRAINING,-0.019,-0.031,0.134,0.098
2022-11-20 11:34:45.082825,CONSENSUS,-0.024,-0.021,0.483,0.055
2022-11-20 11:35:10.894247,TRAINING,-0.022,-0.026,0.548,0.069
2022-11-20 11:35:11.218484,CONSENSUS,-0.021,-0.027,0.226,0.066
2022-11-20 11:35:37.114189,TRAINING,-0.012,-0.048,0.051,0.051
2022-11-20 11:35:37.478274,CONSENSUS,-0.017,-0.036,0.272,0.057
2022-11-20 11:36:03.317306,TRAINING,-0.017,-0.036,0.229,0.049
2022-11-20 11:36:03.635261,CONSENSUS,-0.017,-0.037,0.108,0.032
2022-11-20 11:36:29.613757,TRAINING,-0.017,-0.037,0.008,0.029
2022-11-20 11:36:29.935264,CONSENSUS,-0.015,-0.042,0.084,0.037
2022-11-20 11:36:55.926558,TRAINING,-0.012,-0.048,-0.042,-0.003
2022-11-20 11:36:56.459658,CONSENSUS,-0.013,-0.046,-0.047,0.013
2022-11-20 11:37:26.684286,TRAINING,-0.006,-0.061,-0.07,0.016
2022-11-20 11:37:27.094495,CONSENSUS,-0.011,-0.051,-0.11,-0.006
```

**Figure A.3:** Example of a weight log file

---

## A.4 Epsilon log file

---

The epsilon log files contain the following content everytime the epsilon value changes :

- The time when the epsilon value changed
- The value of the max order known by the neighbour (the inverse of epsilon)

An example of a epsilon log file is presented :

```
time,value
2022-11-20 11:30:47.515560, 2
2022-11-20 11:32:06.763837, 3
2022-11-20 11:32:59.905522, 2
2022-11-20 11:34:45.082825, 3
2022-11-20 11:36:03.317306, 2
```

**Figure A.4:** Example of an epsilon log file



---

## A.5 Training time log file

---

The training time log files contain the following content after each training round :

- The time when the training starts or when the training stops
- A word indicating if the training just started or just stopped

An example of a epsilon log file is presented :

```
time,start_or_stop
2022-11-20 11:30:21.387399,START
2022-11-20 11:30:47.516381,STOP
2022-11-20 11:30:48.077734,START
2022-11-20 11:31:13.926108,STOP
2022-11-20 11:31:14.304751,START
2022-11-20 11:31:40.410062,STOP
2022-11-20 11:31:40.726041,START
2022-11-20 11:32:06.764027,STOP
2022-11-20 11:32:07.288001,START
2022-11-20 11:32:33.167438,STOP
2022-11-20 11:32:33.592605,START
2022-11-20 11:32:59.508734,STOP
2022-11-20 11:32:59.907896,START
2022-11-20 11:33:25.884940,STOP
2022-11-20 11:33:26.198636,START
2022-11-20 11:33:52.134787,STOP
2022-11-20 11:33:52.476969,START
2022-11-20 11:34:18.457971,STOP
2022-11-20 11:34:18.819785,START
2022-11-20 11:34:44.761749,STOP
2022-11-20 11:34:45.085311,START
2022-11-20 11:35:10.894383,STOP
2022-11-20 11:35:11.220685,START
2022-11-20 11:35:37.114323,STOP
2022-11-20 11:35:37.480360,START
2022-11-20 11:36:03.317494,STOP
2022-11-20 11:36:03.638130,START
2022-11-20 11:36:29.614269,STOP
2022-11-20 11:36:29.937368,START
2022-11-20 11:36:55.927257,STOP
2022-11-20 11:36:56.461722,START
2022-11-20 11:37:26.684773,STOP
2022-11-20 11:37:27.099590,START
```

**Figure A.5:** Example of an training time log file



---

APPENDIX B

## Link with Sustainable Development Goals (SDG)

---

Sustainable Development Goals	High	Medium	Low	Non Applicable
SDG 1. No Poverty.			x	
SDG 2. Zero Hunger.			x	
SDG 3. Good Health and Well-being.	x			
SDG 4. Quality Education.		x		
SDG 5. Gender Equality.			x	
SDG 6. Clean Water and Sanitation.		x		
SDG 7. Affordable and Clean Energy.		x		
SDG 8. Decent Work and Economic Growth.			x	
SDG 9. Industry, Innovation and Infrastructure.	x			
SDG 10. Reduced Inequality.			x	
SDG 11. Sustainable Cities and Communities.		x		
SDG 12. Responsible Consumption and Production.			x	
SDG 13. Climate Action.	x			
SDG 14. Life Below Water.	x			
SDG 15. Life on Land.	x			
SDG 16. Peace and Justice Strong Institutions.		x		
SDG 17. Partnerships to achieve the Goal.				x

## B.1 Sustainable Development Goals

---

In 2015, all the United Nations Member States adopted the 2030 Agenda for Sustainable Development<sup>1</sup>. This Agenda is made to help the people and the planet, and also seeks for freedom and peace across the world. It is based on 5 dimensions :

- People : dimension based on ending poverty and hunger, and ensuring that everyone can live in dignity and equality in a healthy environment.
- Planet : dimension based on the protection of the planet, by taking urgent action against climate change, preserve natural resources and encourage sustainable consumption and production.
- Prosperity : dimension based on the fact that everyone should enjoy prosperous and fulfilling lives and that economic, social and technological progress occurs in harmony with nature.
- Peace : dimension based on the fostering of a war-free and violence-free society.
- Partnership : dimension based on a Global Partnership for Sustainable Development, which involves all countries and all people to strengthen global solidarity.

These 5 dimensions encompass 17 SGD<sup>2</sup> (Sustainable Development Goals), which explain more clearly what are the goals and ambitions that people and countries need to aspire to in the coming years. These goals are also divided into 169 targets, that give even more concrete objectives.

## B.2 Relating this study to SGDs

---

Universitat Politècnica de València, as a socially responsible university, has the objective to contribute to these Sustainable Development Goals, and wants to demonstrate how scientific research work can be used in order to achieve these goals. Therefore, this section focuses on explaining the possible uses of this work in order to comply with the UN Sustainable Development Goals.

### B.2.1. Artificial Intelligence and Sustainable Development Goals

First of all, Artificial Intelligence as a whole is bringing a lot of improvements that can be used to make progress on a lot of the SDG described by the UN. Since the work carried out in this study has as objective to create a solution that improves the performance of some current AI solutions, it can therefore be considered as facilitating the use of AI for sustainable development. We will analyze several sectors where AI is used or has the potential to be used, and show their contribution to the SDGs.

---

<sup>1</sup><https://sdgs.un.org/2030agenda>

<sup>2</sup><https://sdgs.un.org/goals>

## Medicine

AI has, for a long time, been used and proven efficient in the field of healthcare and medicine [5]. For example, Verge Genomics<sup>3</sup> is a company that makes use of an AI platform in order to analyze human genomic data and discover new drugs. They have already released a clinically approved drug for amyotrophic lateral sclerosis (ALS), and are working on drugs for diseases such as Parkinson's Disease. Furthermore, AI has been widely used in radiology and has even sometimes proven to present a better percentage of accurate analysis than actual doctors [33].

It is therefore correct to say that AI can (and does) contribute to the second SDG (Good Health and Well-Being).

## Water Crisis

Artificial Intelligence is used in water plants in order to constantly evaluate the quality of the supplied water and to be able to learn from past patterns to detect when something needs to be corrected [26, 3]. Furthermore, AI has been used in marine life as well, in order to distinguish debris from marine animals for instance, or predicting where the big amounts of waste will pile up, or even predicting the amount of marine animals living in a certain radius of a certain point in function of water characteristics [22]. This allows to preserve marine life and reduce marine pollution.

AI can therefore contribute to the following SDGs : Clean Water and Sanitation (SDG 6), Climate Action (SDG 13) and Life Below Water (SDG 14).

## Education

AI is nowadays used in the education field as well in order to improve its quality and the processes involved. Whether it implies the usage of robots or cobots, or a software capable of detecting plagiarism, AI is more than present in this field and is expected to be even more omnipotent in the decades to come [13].

AI therefore contributes to the SDG 4 : Quality Education.

## Justice

AI is even present in the field of justice, which gives place to the concept of predictive justice [6]. The European Council even held several conferences to study the usage of AI software in European justice departments [19].

AI therefore even responds to the SDG 16 : Peace and Justice Strong Institutions.

---

<sup>3</sup><https://www.vergegenomics.com/>

Although this was not an exhaustive list of all sectors where AI can be applied and help towards achieving the 17 SDGs, we showed that AI can unlock a lot of societal, environmental and financial benefits .

### **B.2.2. Federated Learning and Sustainable Development Goals**

The focus of this study was more oriented towards Federated Learning, it is therefore interesting to see if this field, which is a subsection of AI, presents particularities that enhance the accomplishment of the 17 SDGs.

First of all, the Asynchronous Decentralized Federated Learning setting which was the focus of this study allows to get rid of the central server of the centralized architecture. This centralized server consumes a lot of energy, and has to constantly be ready to receive a lot of data and to manipulate it. This proposed decentralized architecture therefore allows not only to be more efficient in certain cases, but also to reduce energy consumption, which aligns with the SDG 12 (Responsible Consumption and Production), and the SDG 13 (Climate Action).

The presented architecture also has the advantage that all participating devices do not need to be identical, which presents practical advantages in some cases. For example, devices such as drones could capture pictures or videos in real time in order to train their local model. In the case of drones, AI could be used to analyze life on land, or even life below water for underwater drones. This unlocks a lot of possibilities in terms of applying AI, and therefore aligns with SDG 14 (Life Below Water) and SDG 15 (Life on Land).

Asynchronous Decentralized Federated Learning has already been put into use in practise in order to detect the faults present in Photovoltaic Stations [41]. This algorithm really suited this solution as the power stations could communicate between them without a central server, and the amount of sent messages per unit of time was quite low so an asynchronous architecture allowed to avoid long waiting times. FL is therefore an asset in order to achieve the SDG 7 (Affordable and Clean Energy) as well as the SDG 13 (Climate Action). Further more, ADFL has been used to predict the production of wind farms in Australia [12]. Every wind farm represents an agent, and communicate between themselves in an asynchronous way in order to predict the electricity production. This project also relates with SDG 7 (Affordable and Clean Energy) as well as with SDG 13 (Climate Action).

### **B.2.3. Dangers of AI**

It is important also to note the possible negative aspects of AI, and how they also might go against SDGs. Indeed, Artificial Intelligence is also widely used as an economic asset, as a tool to improve a company's revenues or profit. Note that this makes AI contribute in some sense to the SDG 9 : Industry, Innovation and Infrastructure. This sometimes leads to big questions regarding the way to treat employees, and the trade-off to be made between the increase in revenues and the well-being of the employees. The progresses made in the fields of robotics and automation in parallel to AI have also made a significant impact on the industry. Indeed, some employees are slowly replaced by machines driven by an AI core. For example, Goldman Sachs, a massive investment bank, em-

ployed 600 traders in their New York City headquarters in 2000 [10]. Today, they employ only 2 traders, the rest of the operations being conducted by machines running automated trading programs making use of various AI techniques [10]. This example directly impacts the following SDGs :

- SDG 1. No Poverty : the unemployment of a large number of workers gives place to more poverty
- SDG 8. Decent Work and Economic Growth : the unemployment of a large number of workers goes against the principle of providing a decent work for all
- SDG 10. Reduced Inequality : the use of AI increases inequalities in this case as the owners make a lot of profit out of it whereas a large amount of workers are left without a salary

It is important to be conscious about these risks when making use of AI, and many experts warn about this. For example, In January 2015, Stephen Hawking, Elon Musk and dozens of other artificial intelligence experts published an open letter calling for research on the societal impacts and the possible negative consequences of AI [48].