UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Estudio de la utilización de coaliciones en algoritmos de aprendizaje federado en sistemas multi-agente

Trabajo Fin de Máster

Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

AUTOR/A: Enguix Andrés, Francisco

Tutor/a: Carrascosa Casamayor, Carlos

Cotutor/a: Rincón Arango, Jaime Andrés

CURSO ACADÉMICO: 2022/2023

Resum

Una de les últimes grans evolucions en l'aprenentatge distribuït ha estat el desenvolupament de l'aprenentatge federat. Aquest tipus d'aprenentatge s'ha posat en pràctica amb èxit utilitzant, entre d'altres, sistemes multiagent. El treball que es planteja aquí és l'estudi de l'evolució d'aquest tipus d'algoritmes quan es defineixen grups o coalicions entre aquests agents que estan realitzant l'aprenentatge, de manera que els agents que pertanyen a la mateixa coalició són més afins que aquells que no són del mateix grup. Aquest treball plantejarà aquest estudi a nivell teòric i pràctic, utilitzant per a aquest últim cas l'eina de desenvolupament de simulacions FIVE i agents SPADE.

Paraules clau: coalicions, aprenentatge federat, five, simulador, entorn virtual, ive, mas, agents

Resumen

Una de las últimas grandes evoluciones en el aprendizaje distribuido ha sido el desarrollo del aprendizaje federado. Este tipo de aprendizaje se ha llevado a la práctica con éxito utilizando, entre otros, sistemas multi-agente. El trabajo que se plantea aquí es el estudio de la evolución de este tipo de algoritmos cuando se definen grupos o coaliciones entre esos agentes que están realizando el aprendizaje, de forma que los agentes que pertenecen a la misma coalición son más afines que los que no son del mismo grupo. Este trabajo planteará este estudio a nivel teórico y práctico, usando para éste último caso la herramienta de desarrollo de simulaciones FIVE y agentes SPADE.

Palabras clave: coaliciones, aprendizaje federado, five, simulador, entorno virtual, ive, mas, agentes

Abstract

One of the latest significant advancements in distributed learning has been the development of federated learning. This type of learning has been successfully implemented using, among other things, multi-agent systems. The work proposed here is the study of the evolution of this type of algorithm when groups or coalitions are defined among the agents that are performing the learning, so that agents belonging to the same coalition are more aligned than those who are not in the same group. This work will propose this study at a theoretical and practical level, using the simulation development tool FIVE and SPADE agents for the latter case.

Key words: coalitions, federated learning, five, simulator, virtual environment, ive, mas, agents

Contents

List of Figures

CHAPTER 1
Introduction

In the realm of distributed learning, the concept of coalitions has arisen as a powerful tool to improve the efficiency and effectiveness of learning processes. A coalition, in the context of multi-agent systems, is a group of agents that collaborate to achieve a common goal. This collaboration can be based on complementary skills or utilities, and the formation of coalitions takes into account various factors, including the availability of information, the presence of externalities, the position of the agents and the concept of trust and reputation among agents.

This thesis proposes an approach that builds upon the principles of Asynchronous Co-Learning (Asynchronous CoL or ACoL), a consensus-based decentralized federated learning algorithm, but introduces the concept of coalitions to study the learning process. Specifically, in this approach, an agent increases the probability of sending a message to the agents that belong to its coalition, thereby strengthening intra-coalition communication and learning.

The advantages of forming coalitions in distributed learning are manifold. Firstly, coalitions allow for the pooling of resources and knowledge, leading to more efficient learning processes. Secondly, coalitions can provide a platform for agents to build trust and reputation, which are crucial for effective collaboration in multi-agent systems. Thirdly, by focusing on communication within coalitions, we can potentially reduce communication overhead and improve the scalability of the learning process.

The proposal is particularly interesting due to the dynamic and distributed nature of the learning process. In such a setting, the formation of coalitions can provide a structured approach to manage the complexity of the learning process, while still maintaining the flexibility and robustness of asynchronous communication. Furthermore, by enhancing intra-coalition communication, we can potentially accelerate the convergence of the learning process and improve the quality of the learned models in certain scenarios.

1.1 Motivation

The motivation for undertaking this project primarily arises from the acceptance of the *Formación Personal Investigador* (FPI) pre-doctoral contracts for doctoral candidates grant for the COordinated intelligent Services for Adaptive Smart areaS (COSASS) project [1]. This project aims to dive into the issue of distributed coordination and decision-making in challenging environments using heterogeneous physical Internet of Things (IoT) devices operating on the edge. These devices carry out specific tasks on behalf of independent service providers or producers and need to collaborate and coordinate, sharing their

[1] https://cosass.usal.es/

resources and capacities, to achieve their goals more efficiently. The author's passion for multi-agent system research also fuels this project. While pursuing the Master's Degree in Artificial Intelligence, Pattern Recognition and Digital Imaging (MIARFID), the author contributed to the development of two papers: "GTG-CoL: A new Decentralized Federated Learning based on Consensus for dynamic networks" [1] submitted to PAAMS [2] and "Consensus-based Learning for MAS: Definition, implementation and integration in IVEs" sent to IJIMAI [3]. The former has been accepted, and the latter is currently under review.

1.2 Objectives

The primary objective of this academic work is to study the use of coalitions in federated learning algorithms within the context of multi-agent systems. To achieve this, we will address the following secondary objectives:

- Improve the FIVE framework to enable the use of this three-dimensional simulation framework to form dynamic groups of intelligent agents, based on their positions. This includes the introduction of the concept of artifacts, which are entities that can be used by agents and add richness to the simulations.
- Enhance the ACoL framework to test distributed federated learning algorithms based on consensus and coalitions. This includes studying the current limitations that prevent it from functioning with medium-sized neural networks and finding solutions to these issues.
- Design and implement a new algorithm that introduces coalitions to asynchronous distributed federated learning, and conduct tests with it.
- Integrate neural models into IoT devices and optimize their memory space and incorporate them into FIVE for use in simulations. This objective is related to the COSASS project described in section 1.1.

1.3 Memory structure

The structure of this thesis has been designed to build upon a knowledge base that allows us to understand the operation and purpose of all the components that make up this academic work. Thus, a comprehensive state-of-the-art review has also been included to understand the context of this work.

The first chapter serves as an introduction to the academic work, providing a broad overview of its content without delving into specifics. It also includes the motivation that encourages the study of coalitions and the improvement of the FIVE framework, which originated as the Simulator of Adaptive Smart Areas based on SPADE Multi-Agent Systems (SASAMAS) [2] in the final degree project.

The second chapter, titled *State of the Art* contains the necessary definitions to understand the theoretical concepts and technologies that underpin this work. It provides basic definitions of the research line of Multi-Agent Systems, as well as the fundamental algorithms upon which other algorithms we discuss are based, their interrelationships, and opportunities for innovation.

[2] https://www.paams.net/
[3] https://www.ijimai.org

The third chapter, *Proposal*, offers a detailed study of the opportunities for improvement in the frameworks mentioned above, as well as the functionalities and properties that the new algorithm to be implemented should have. It concludes with a presentation of the studies that will be conducted to test the algorithm that works with mobile agents.

The fourth chapter, *Solution Design*, involves implementing all the proposals detailed in the previous chapter. This allows us to elaborate and, in some cases, show small code snippets to demonstrate the specific operation of the solutions we have designed and implemented.

The fifth chapter, *Case of Study*, consists of various studies conducted on both the developed software and the implemented algorithms. It involves using the tools developed in the previous chapter to conclude certain scenarios.

Finally, we will complete the work by synthesizing it, and discussing whether we have managed to address the proposed objectives. Likewise, we will also discuss future work and the open lines of research that this academic work presents.

CHAPTER 2
State of the art

2.1 Definitions

Intelligent agents An intelligent agent is a system that perceives its environment and acts upon that environment in order to achieve specific goals. They also can collaborate with other agents and share information. [3]. Key characteristics of intelligent agents include:

- Social ability: Intelligent agents can interact with other agents or humans in their environment to achieve their goals.
- Autonomy: Intelligent agents operate without human intervention, and they can control their actions and internal state.
- Reactivity: Intelligent agents perceive their environment and respond on time to changes that occur in it.
- Adaptability: Intelligent agents learn from their experiences and can adapt their behaviour based on their learning.
- Pro-activeness: Intelligent agents do not simply act in response to their environment; they are capable of taking the initiative, such as exhibiting goal-directed behaviour by taking the initiative.
- Goal-oriented: Intelligent agents are designed to fulfil certain objectives or tasks. They use their intelligence to make decisions that maximize their chances of achieving these goals.

Artifacts An artifact is a computational entity representing a tool or a resource within the environment where agents are situated. In contrast to intelligent agents, artifacts do not perform autonomous actions to achieve goals. However, they offer functionalities that agents can use during runtime. Artifacts are considered first-class entities of the environment, and they play a crucial role in supporting individual and collective activities of the agents. [4]

Multi-agent systems A Multi-Agent System (MAS) is composed of multiple interacting intelligent agents. These systems can be used to solve problems that are difficult or impossible for an individual agent or a monolithic system to solve. In a MAS, agents work together by communicating and coordinating their actions. MAS can be used to model and understand complex systems, such as social networks, ecosystems, or the economy.

They are also used in practical applications such as distributed computing, logistics, or autonomous vehicles. [5] Key characteristics of MAS include:

- Communication: Agents communicate with each other to coordinate their actions, share information, or negotiate.
- Decentralization: There is no single controlling agent. Each agent can operate independently and makes its own decisions.
- Cooperation or Competition: Depending on the system, agents might cooperate to achieve a common goal, or they might compete against each other to achieve their individual goals.
- Adaptability: MAS can adapt to changes in the environment. If one agent fails, others can take over its tasks.
- Complexity: MAS can model complex, dynamic environments. The behaviour of the system emerges from the interactions of the individual agents.
- Diversity: Agents in a MAS can be heterogeneous, meaning they can have different capabilities, knowledge, or goals. This diversity can make the system more robust and flexible.

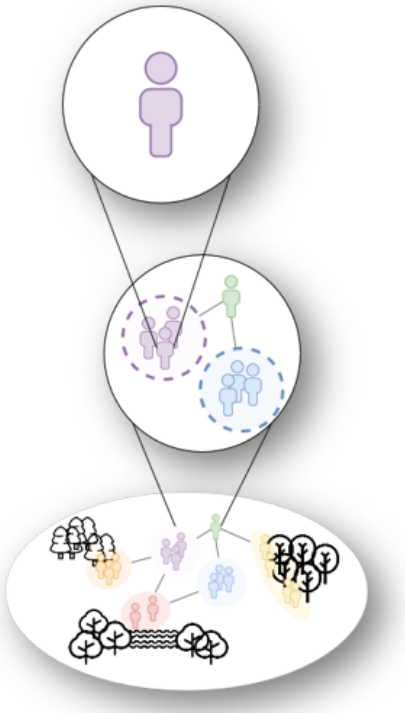


Figure 2.1: Scheme of the concept of Intelligent agent, MAS and IVE.

Intelligent Virtual Environment An Intelligent Virtual Environment (IVE) is a digital simulation or 3D virtual world that incorporates artificial intelligence inhabiting intelligent agents, where they can interact and their behaviour can be easily validated. [6] IVEs can simulate complex and dynamic environments. Intelligent agents can interact with each other and with the environment in complex ways, leading to emergent behaviours that can make the simulation more realistic and interesting for researchers.

Coalition A coalition is a group or alliance formed by intelligent agents that come together to achieve a common goal or perform a specific task. It involves the cooperation and coordination of individual agents, each possessing its capabilities and resources, to accomplish objectives that may be beyond the capacity of a single agent. The agents in a coalition typically share information, exchange messages, and collaborate in decision-making processes to maximize their collective effectiveness. Forming coalitions in MAS allows for distributed problem-solving, improved efficiency, and the ability to dive into complex tasks through collective intelligence and synergy. [7]

2.2 XMPP

XMPP (Extensible Messaging and Presence Protocol) [1] is a robust and widely adopted protocol for instant messaging and presence information exchange. It is based on the Extensible Markup Language (XML) and operates at the application layer of the network stack.

One of the critical characteristics of XMPP is its openness and extensibility. Unlike proprietary messaging protocols, it is an open standard that allows for the developing of additional features and extensions. This flexibility enables XMPP to adapt to various communication requirements and scenarios.

XMPP supports secure communication by integrating encryption mechanisms such as SASL (Simple Authentication and Security Layer) and TLS (Transport Layer Security). These protocols ensure the confidentiality, integrity, and authenticity of the exchanged messages, providing a secure environment for communication between agents.

Another notable aspect is its decentralized nature. The networks consist of interconnected servers that facilitate communication between agents. This decentralization allows organizations and individuals to set up their own XMPP servers, giving them control over their messaging infrastructure and promoting autonomy and privacy.

By utilizing XMPP, intelligent agents can establish reliable and efficient communication channels, providing a range of features and functionalities, including real-time messaging, presence information, and data synchronization. Agents can exchange messages, share updates on their availability, and coordinate their actions in a distributed and collaborative manner.

2.3 SPADE

SPADE (Smart Python Agent Development Environment) [8] is a comprehensive framework designed for developing intelligent agents using the Python programming language. One of the distinctive features of SPADE is its integration of the XMPP (Extensible Messaging and Presence Protocol) as the underlying instant messaging protocol.

Effective communication plays a vital role in multi-agent systems, and SPADE offers a convenient solution by incorporating a built-in message dispatcher. This dispatcher facilitates seamless communication between agents, allowing them to exchange information, coordinate actions, and collaborate towards achieving their individual or collective goals.

The core concept in the SPADE agent model revolves around behaviours. Behaviours are specific tasks that agents repeatedly perform, following predefined patterns of time.

[1] https://xmpp.org/

SPADE supports various behaviours, including one-shot behaviours that execute once, periodic behaviours that repeat at regular intervals, and FSM (Finite State Machine) behaviours that exhibit state-dependent actions. Additionally, SPADE introduces BDI (Belief Desire Intention) [9] behaviour, enabling agents to exhibit reactive and deliberative behaviours based on their beliefs, desires, and intentions.

The popularity of SPADE has been steadily growing, as evidenced by its substantial download count. In January 2020 alone, SPADE recorded 369 downloads [8]. This trend signifies the increasing recognition and adoption of SPADE as a powerful tool for building intelligent agent systems. The third version of SPADE underwent significant project restructuring [8], resulting in a more robust and formally designed system. This evolution has made SPADE an even more reliable and efficient framework for programming intelligent agents.

2.4 FIVE: Flexible Intelligent Virtual Environment designer

The Flexible Intelligent Virtual Environment designer (FIVE) framework, previously called Simulator of Adaptive Smart Areas based on SPADE Multi-Agent Systems (SASAMAS) [2], is a comprehensive system designed to facilitate the testing of multi-agent systems in a preliminary stage before deploying experiments in the real environment. FIVE is designed to be user-friendly, allowing for easy and quick testing of multi-agent systems. This is achieved through a text-based editable simulator and a system that allows to establish the behaviour of different intelligent agents easily, as it shows Figure 2.2.

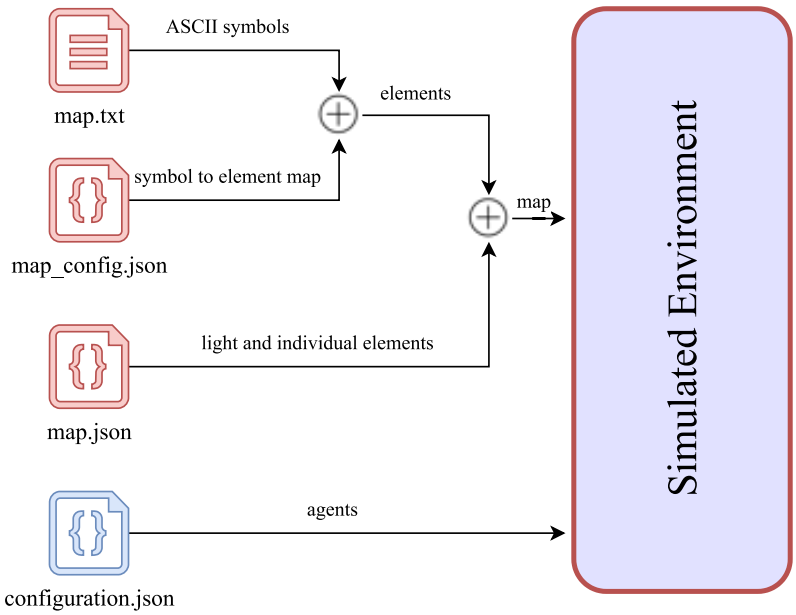


Figure 2.2: Simulator environment and agent generation from input files. The first three red files generate the intelligent virtual environment (composed of light objects, agent spawn points, and other elements), and the last blue file is used to generate the agents.

It is composed of two parts that we are going to name FIVE server and FIVE client. The FIVE server, which is the part that manages the three-dimensional representation of the IVE and serves the avatars of the intelligent agents in the simulation. It was developed using Unity, which is a powerful graphics engine primarily designed to create two-dimensional and three-dimensional video games. However, its applications extend

beyond gaming and are increasingly utilized in diverse industries such as film production, architecture, aerospace, and vehicle design. [10] [11]

On the other hand, the FIVE client creates an abstraction layer between the user and SPADE. It consists of a system that allows programming the behaviour of different intelligent agents, just filling a template with the states of the agent's Finite State Machine (FSM), as Figure 2.3 shows. Also, there is an API named `Commander` that allows for quick and easy programming of agents and network communication with the server. It also allows for instructions to be given to the agent's avatar in the FIVE server.

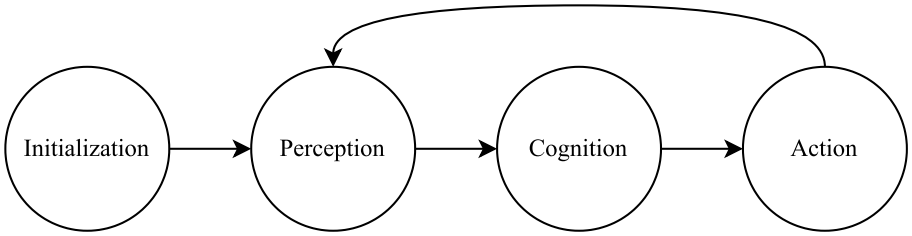


Figure 2.3: FIVE agent FSM to control the avatar in the IVE.

FIVE, which is easily editable, simulates a three-dimensional environment that allows interaction with intelligent agents inhabiting this environment. It also allows for the loading of textures of elements from a path to a directory with images in runtime. This process is optimized to minimize the scene loading time, adding agility and flexibility to the experiments. It has a system that lets agents communicate via the network with the simulator, and among themselves, so researchers can collaborate remotely from different geographical locations worldwide.

FIVE grants the quick and easy creation of different types of IVEs, as it is shown in Figure 2.4. It also allows for easy modification of these environments, not only changing the arrangement of elements and the number of agents in the scene but also the environmental conditions of the simulation. It saves time and resources by allowing for the testing of multi-agent algorithms in a virtual environment before deploying them in the real world. This early detection of anomalies or failures can save the cost of human and material resources caused by the deployment of the experiment in the real world.

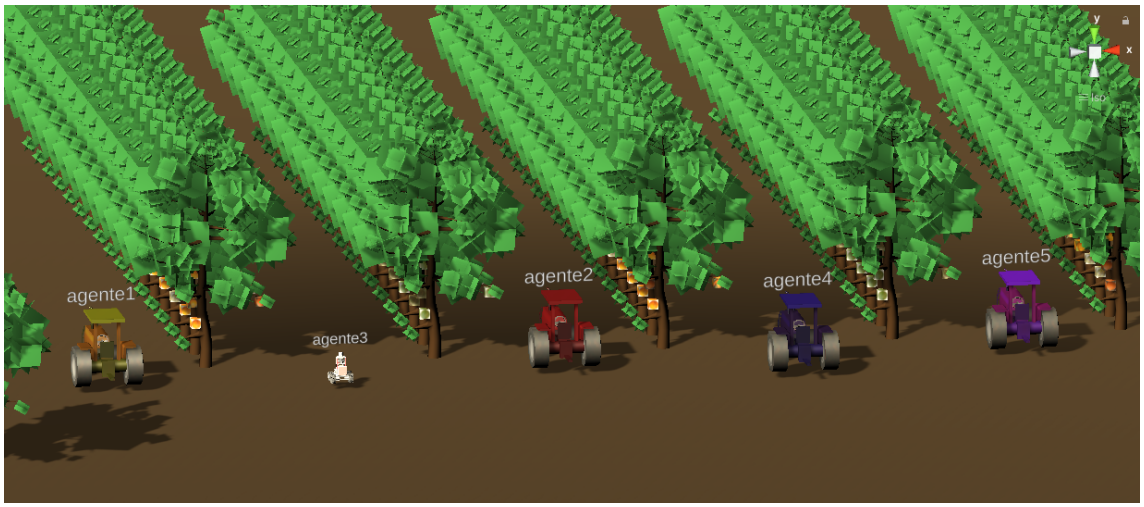


Figure 2.4: FIVE agent FSM to control the avatar in the IVE.

2.5 Federated learning

Federated Learning (FL) [12] is a machine learning approach that allows a model to be trained across multiple decentralized devices or servers holding local data samples, without exchanging them. This approach is particularly useful in scenarios where data privacy is critical. It has gained significant attention due to its potential to improve the performance of Machine Learning (ML) models by distributing the training process across multiple clients and combining the models in a central server.

Federated Learning is a paradigm shift from the traditional centralized ML model, where all the data needed to train a model is usually collected and processed in a single, central location. Instead, in FL, the data remains on the local devices (like a personal computer or a business server), and only model weight updates are shared and aggregated to form a global model. This approach allows for creating models without the need to share raw data.

This approach is used in various applications where data privacy and security are paramount. The advantages of FL are numerous. First and foremost, it addresses privacy concerns by keeping sensitive data on the original device, reducing the risk of data breaches. It also allows for more personalized models, as each local model can learn from data that is highly relevant to the individual user. Furthermore, by reducing the need to transmit large amounts of data, Federated Learning can save bandwidth and reduce latency.

It is particularly useful for intelligent agents and MAS. In these systems, each agent can be seen as a node in a FL network, learning from its local observations and experiences. This allows the agents to learn more effectively in environments with limited observations or where the environment is non-stationary (i.e., it changes over time).

In a MAS, FL allows the agents to collectively learn a model that is more robust and generalizable, as it is trained on a diverse set of data from different agents. This can lead to improved performance and efficiency in the system as a whole. Moreover, in a MAS where each agent has its own private data, FL allows the agents to learn from the collective data without compromising their data privacy. This is particularly important in scenarios where the agents are owned by different entities that do not wish to share their data.

2.6 Consensus

Consensus refers to achieving a unified value or agreement on overall intelligent behaviour and actions through local or regional communication and cooperation between individual agents. It has gathered significant attention from scientific communities and many algorithms have been developed to ensure that agents converge to a consensus value. [13] [14]

Consensus algorithms like the one proposed by Olfati-Saber and Murray [13] are used in distributed systems to achieve agreement on a single data value among distributed processes or systems. They are often used in scenarios where a group of agents, each with its own private estimate of some common parameter, seek to reach a consensus on the parameter's value through local interactions.

While both Federated Learning and consensus algorithms involve distributed computation and communication between multiple agents or nodes, they are used for different purposes and operate differently. FL is specifically designed for training machine

learning models in a decentralized and privacy-preserving manner, while consensus algorithms are more general-purpose tools for achieving agreement in distributed systems.

That being said, some federated learning algorithms do incorporate elements of consensus algorithms to ensure that the local models converge to a global model. [15] However, this is not a fundamental part of the FL framework and is not used in all federated learning algorithms.

We will use the consensus algorithm as a problem where agents are connected by a communication network that we will model as an undirected graph, and they agree on a variable of interest, which will be each of the weights of the matrices of the neural network that each agent has. Formally, we define the communication graph of the agents as $G = (A, E)$, where A is the set of vertices whose nodes represent the agents and $E \subseteq A \times A$ is the set of edges of the graph that represents the communication capacity of two agents (a_i, a_j), denoting that agent a_i is connected with agent a_j. Therefore, we can define the neighbourhood of an agent a_i as $N_i = a_j \in A : (a_i, a_j) \in E$. We will denote the weight matrix of an agent with W_i and it will be trained with a training dataset tr_i that can be different for each agent.

$$W_i = (W_{i,1}, \ldots, W_{i,k}) \tag{2.1}$$

where $W_{i,j} \in \mathbb{R}^{n,m}$ represent the weights learned by agent a_i for layer j of its neural network. If we also consider the consensus process in MAS as an iterative process, agent a_i calculates the new value $x_i(t+1)$ of each iteration as follows:

$$x_i(t+1) = x_i(t) + \varepsilon \sum_{a_j \in N_i} [x_j(t) - x_i(t)] \tag{2.2}$$

where ε is the learning step, which is a factor limited by the maximum degree of the graph representing the agent communication network. The algorithm converges to the average of the initial values as long as $\varepsilon \leq \frac{1}{\max d_i}$, where d_i denotes the number of neighbouring agents that agent a_i has, i.e., $|N_i|$. Therefore, equation 2.1 can be represented by:

$$W_i(t+1) = W_i(t) + \varepsilon \sum_{a_j \in N_i} [W_j(t) - W_i(t)] \tag{2.3}$$

Finally, we will reach the consensus of the weights when we reach a value $W^\star$, such that $W_i = W^\star, \forall i \leq |A|$, that is, all agents have the same weight value.

2.7 FLaMAS

FLaMAS (Federated Learning based on Multi-Agent Systems) [16] is a framework based on SPADE agents, that combines the power of deep learning and MAS to create a distributed learning tool. Developed by Dr. Jaime Rincón, Dr. Vicente Julián, and Dr. Carlos Carrascosa, FLaMAS is designed to train deep learning models, adhering to the principles of FL. It ensures the privacy of the data, as only the model being learned is shared, not the data used for training.

In the FLaMAS system, there are two distinct agent roles: the Client Role and the Server Role. The Client's primary function is to receive the input data from the device where it is embedded, train the model, and send it to the agent playing the Server role. The Server, on the other hand, is responsible for receiving the different trained models

from the Client agents at each iteration. The Server agent averages all these models, and this new unified model is sent to the different Client agents, as Figure 2.5 shows.

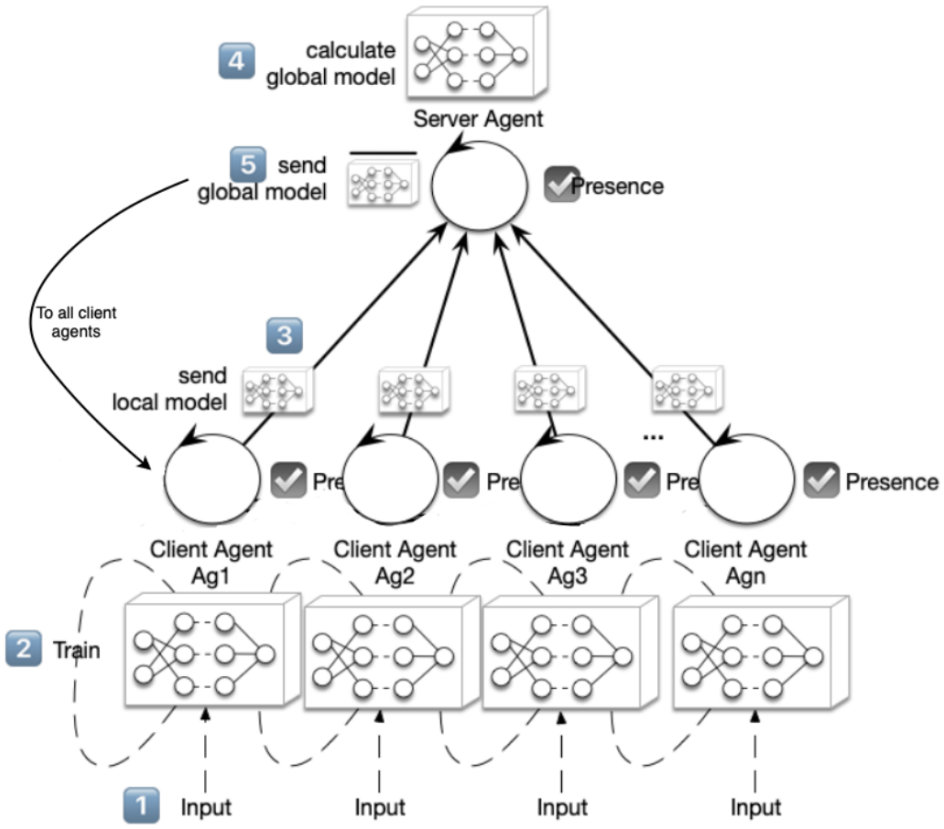


Figure 2.5: FLaMAS framework scheme.

FLaMAS has been evaluated with different experiments, and the results obtained demonstrate the benefits of combining the federated system with SPADE agents, while facilitating flexibility and scalability. [16]

2.8 Consensus-based Learning

Consensus-based Learning, also known as Co-Learning or CoL, is a new multi-agent learning algorithm built upon consensus [13], and it was developed in collaboration with Dr. C. Carrascosa, Dr. M. Rebollo and Dr. J. A. Rincón. The paper has been submitted to IJIMAI [2] and it is currently undergoing revision. CoL was designed to manage some of the restrictions of FL algorithm. While FL has been instrumental in advancing distributed learning, it has particular limitations, such as the need for synchronization and the lack of failure tolerance.

Co-Learning, on the other hand, is a fully decentralized approach to distributed learning. It is conceived to work in environments with a high probability of communication failure, where agents communicate sporadically, or when they must deal with disconnection periods to save battery. This is especially useful in rural areas, characterized by limited connectivity, where the system may remain isolated without supervision for extended periods.

[2] https://www.ijimai.org

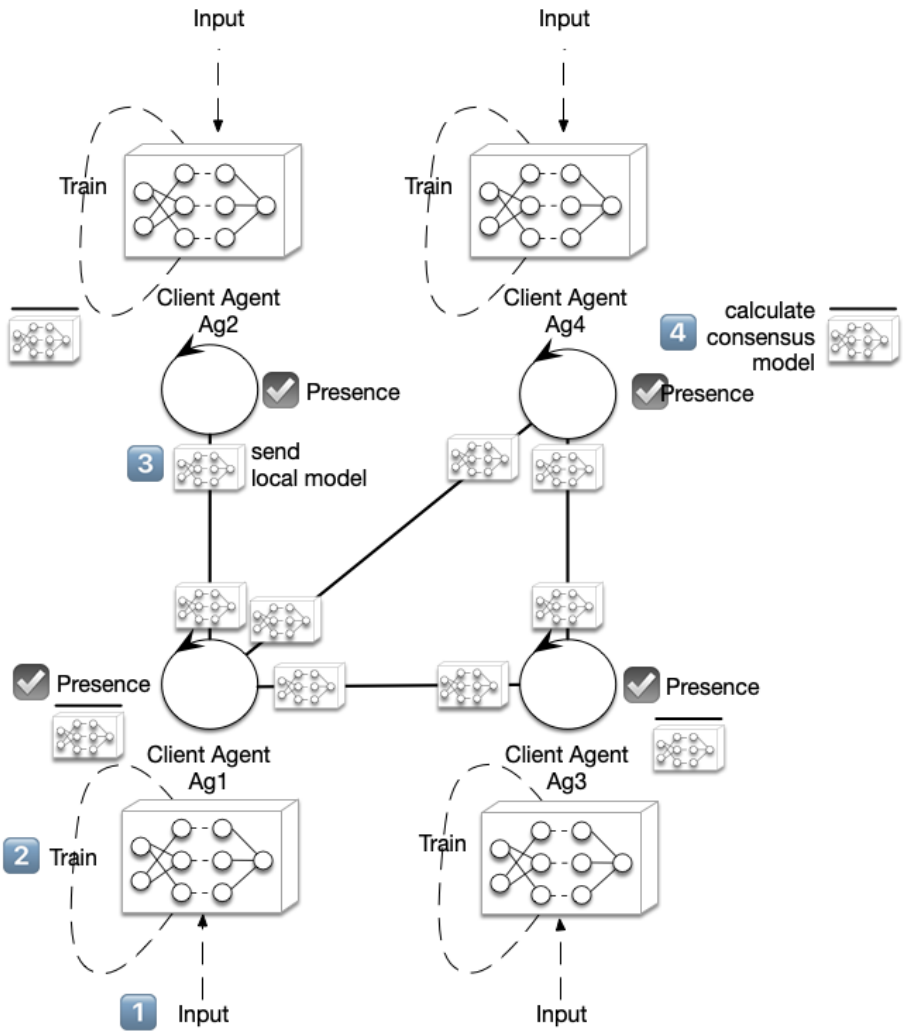


Figure 2.6: Scheme of four SPADE agents during CoL algorithm.

In the CoL approach, each agent in a MAS learns and shares its model with its local neighbours. The learning process is iterative, with each agent calculating a new value in each iteration based on the models shared by its neighbours. This process converges to an averaged model, similar to how a central server would in the centralized approach.

One of the key advantages of Co-Learning is that it protects the privacy of the data used for the learning process by each agent in its local learning. This feature is also present in FL, but CoL takes it a step further by eliminating the need for a central server. CoL has been implemented using SPADE agents, and it has been tested in a simulation of an orange orchard field using the FIVE framework, demonstrating the practical application of Co-Learning in real-world scenarios.

The Consensus-based Learning Algorithm can be described as a set of agents learning a model through a neural network, where all the agents share the same neural network structure. This allows sharing the model learned by each agent with its local neighbours and making a consensus of such model based on the equation 2.3. This model is formed by the weight matrices described by the equation 2.1, which are the result of the training learning process. This consensual model is used for each agent in the next training process.

Figure 2.6 shows the Co-Learning algorithm in a network formed by four SPADE agents. Each agent a_i following the CoL algorithm (Algorithm 2.1) will make e epochs

of training. The result of this training is a set of k matrices of weights, as we show in equation 2.1. Then, c iterations of the consensus algorithm will be performed, following the equation 2.3, leading to a new set of k matrices that will be used in the training process again.

Algorithm 2.1 $CoL_{a_i}(e, k, c)$ - Co-Learning Algorithm for agent a_i

```
while !doomsday do
    for f ← 1, e do
        W ← Train(f)
    end for
    for j ← 1, k do
        x_i(0) ← W_j
        for t ← 1, c do
            x_i(t+1) ← x_i(t) + ε Σ_{a_j ∈ N_i} [x_j(t) − x_i(t)]
        end for
    end for
end while
```

The process is executed in parallel as many times as parameters the neural network has. It can be considered a vectorized version of the evolution that Figure 2.7 shows, where four SPADE agents with initial values $x(0) = \{0.2, 0.4, 0.6, 0.8\}$ reach a consensus, so the $\langle x(0) \rangle = 0.5$.

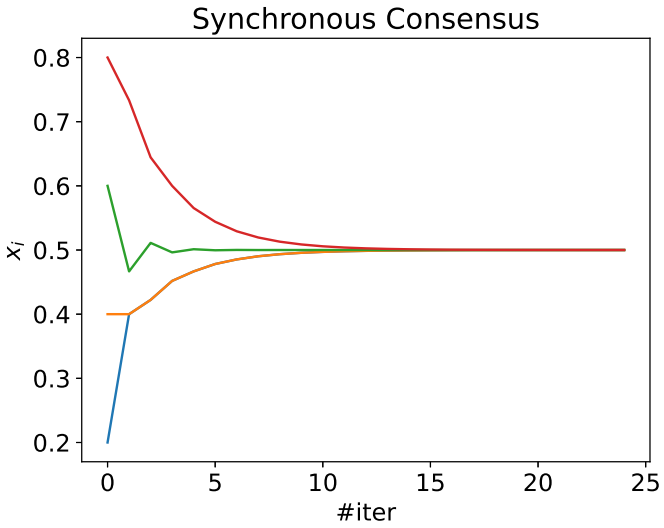


Figure 2.7: Consensus evolution of four SPADE agents.

2.9 Asynchronous Co-Learning

Asynchronous Co-Learning (Asynchronous CoL or ACoL) is an innovative approach based on Co-Learning to distributed learning, that has been developed to handle the challenges of real-world systems where agents may not always be able to communicate synchronously. This approach is particularly appropriate in environments where agents may experience unsteady connectivity or where they perform on different schedules.

Asynchronous CoL builds upon the principles of Co-Learning, but introduces the ability for agents to perform and communicate asynchronously. In this approach, each

agent independently trains a model and shares its parameters with a random neighbour whenever it is possible. This allows for a more flexible and robust learning process that can adapt to the dynamic nature of real-world environments. It has been shown to offer several advantages over synchronous methods. For instance, if one device is down or takes too long, the other devices can still communicate and update their models. This makes the learning process more resilient to delays and disorders.

The algorithm that defines ACoL is shown in 2.2. The parameters are the same as the CoL algorithm shown in 2.1. The e epochs of training are the number of epochs that the agent a_i performs before sending the weights. The result of this training is a set of k matrices of weights, and then c iterations of the consensus algorithm will be performed. It is critical to note that the weights are sent to a random neighbour in the $RANDOM\_SELECT(N_i)$ step.

Algorithm 2.2 $ACoL_{a_i}(e, k, c)$ - Asynchronous Co-Learning algorithm for agent a_i

while !doomsday **do**
  for $f \leftarrow 1, e$ **do**
    $W \leftarrow Train(f)$
  end for
  for $t \leftarrow 1, c$ **do**
    $x_j = RANDOM\_SELECT(N_i)$
    for $j \leftarrow 1, k$ **do**
      $x_i^t \leftarrow W_j$
      $x_i^{t+1} \leftarrow (1 - \varepsilon)x_i^t + \varepsilon x_j^t$
      $W_j \leftarrow X_i^{t+1}$
    end for
  end for
end while

CHAPTER 3
Proposal

3.1 Improvements to the FIVE framework

3.1.1. Motivation

The need for improvement of the FIVE framework is driven by the acceptance of the *Formación Personal Investigador* (FPI) pre-doctoral contracts for the training of doctoral candidates grant for the COordinated intelligent Services for Adaptive Smart areaS (COSASS) project [1]. This project aims to address the issue of distributed coordination and decision-making in challenging environments using heterogeneous physical Internet of Things (IoT) devices operating on the edge. These devices perform specific tasks on behalf of independent service providers or producers and need to collaborate and coordinate, sharing their resources and capacities, in order to achieve their goals more efficiently. Additionally, the motivation to enhance the framework's capabilities is also driven by the need to continue a project that provides MAS researchers with a versatile tool for conducting experiments in a controlled three-dimensional environment that is easily constructed and modified. As a result of the described improvements in this section, two scientific articles titled "GTG-CoL: A new Decentralized Federated Learning based on Consensus for dynamic networks" [1] sent to PAAMS [2] and "Consensus-based Learning for MAS: Definition, implementation and integration in IVEs" sent to IJIMAI [3] have been submitted, with the former being accepted and the latter currently undergoing review.

3.1.2. Agent Behavior Templates and Algorithm Plug-in System

The primary objective of the FIVE framework is to enhance the speed and versatility of conducting scientific studies in the field of multi-agent systems. Given this, we can consider what additional improvements might facilitate the programming of SPADE agents, for which we already have an abstraction layer in FIVE. This abstraction layer is composed of a template where we write the behavior of the agents, but there is room for improvement in complex algorithms where multiple agent roles exist. An example of this could be FL-based algorithms, where there are Client and Server roles. Currently, this can be addressed in FIVE in two ways:

1. We create a behavior template for all agents and distinguish between agents with the Client role and those with the Server role within the template code.

[1] https://cosass.usal.es/
[2] https://www.paams.net/
[3] https://www.ijimai.org

2. We create two instances of the FIVE client and complete both behavior templates independently. In one instance, we complete the behavior template of the agent with the Server role, and in the other instance, we complete the Client agent's behavior.

Both approaches are perfectly valid and functional, but it would be simpler if we could establish the behavior template for each agent within a single instance of the FIVE client, through the agents' configuration file `configuration.json`. Additionally, it would be advantageous to use a plug-in system for complex algorithms, such as the FL-based algorithm: FLaMAS [12][16]. This plug-in system would significantly enhance the utility of FIVE if it allowed the import of generic code from a complex algorithm, simply by adding the algorithm's name to a list of imported algorithms, and then allowed us to program the specific part of them in the same agent behavior template. For example, we could add the term "flamas" to a list of `algorithms` in the `configuration.json` file, which would cause the FIVE client to load the generic behavior of FLaMAS into the SPADE agent, and we would only program its specific methods in the `EntityShell` behavior template of the agent.

3.1.3. Optimizing FIVE Networking System while Improving Capabilities

Communication management with agents is one of the most important features of the FIVE framework, as communications are also one of the main attributes of intelligent agents. Agents need to communicate with each other to cooperate, share resources, assign tasks, and organize themselves. For this reason, we should consider if there is a way to improve the current network communication system of the agents.

Currently, intelligent agents are managed by FIVE through network sockets. Each agent has direct communication with the FIVE server through two sockets, one used for command sending and another for image sending, and they have been carefully programmed to avoid race conditions due to the necessary thread management to prevent blockages. This system is efficient and robust, but if we analyze it in depth, we can change the communication structure to relieve the computational load from the FIVE server, as the number of sockets the server must attend to scales linearly with the number of intelligent agents connecting to it, as the Figure 3.1 shows.

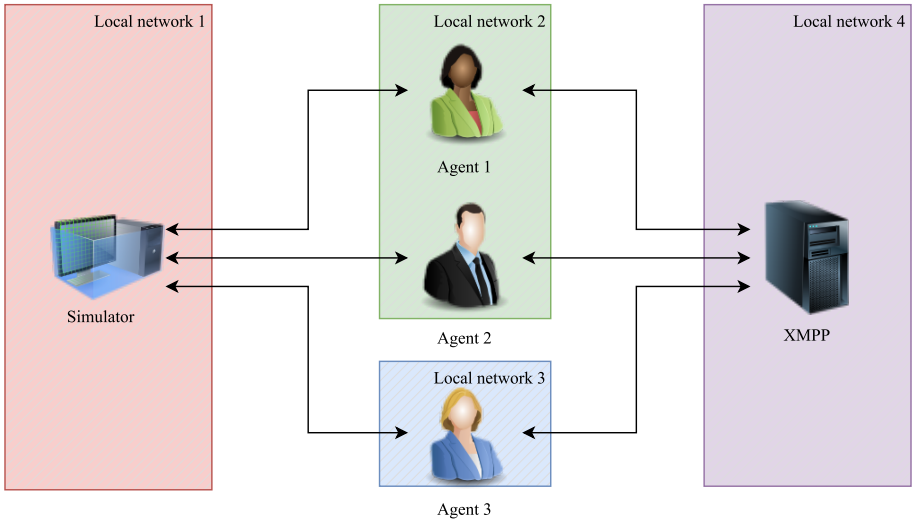


Figure 3.1: Scheme of the original FIVE architecture.

The improvement we propose in this section consists of leveraging the XMPP server required by the FIVE client agents, as this XMPP server is an indispensable requirement

in our system, because FIVE agents are based on SPADE agents. The FIVE server could communicate only with the XMPP server. In this way, the number of network sockets is constant in relation to the number of agents instantiated in our IVE, as the Figure 3.2 shows. Providing not only greater computing capacity to the FIVE server by relieving the socket management load, but also new features derived from integration with the XMPP protocol.

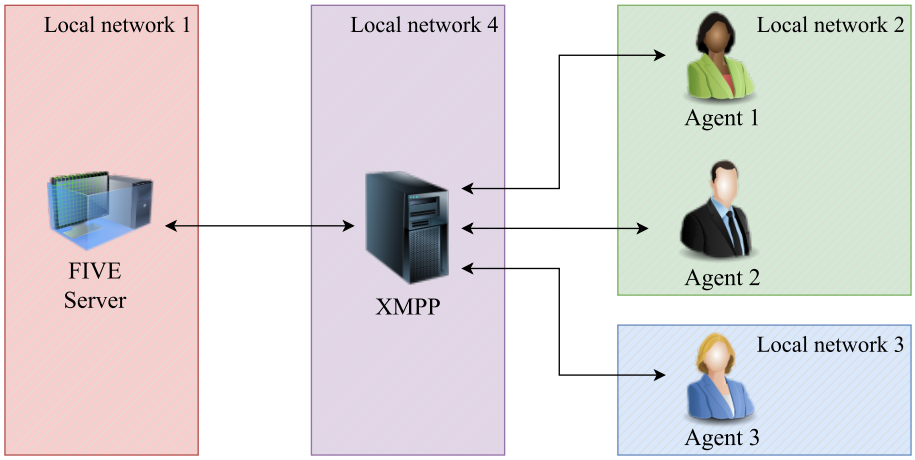


Figure 3.2: Scheme of the new proposed FIVE architecture.

The integration of the FIVE server with the XMPP protocol would turn the FIVE server into a piece of software with agent-like qualities. It would have access to presence, so agents would know when the server is connected and operational, and vice versa. On the other hand, intelligent agents could also subscribe to it and use semantically rich communication, thanks to the metadata of XMPP messages.

3.1.4. Including Artifacts into the IVE

The simulation of intelligent agents in an IVE is especially useful for studying complex and dynamic systems, with task assignment among agents and resource sharing. In MAS, there is the concept of an artifact, which is defined as a computational entity or object that exists within the environment and is used by agents to interact and exchange information with each other. Artifacts can be considered as shared resources or communication channels that facilitate communication and coordination among intelligent agents. Moreover, artifacts are closely related to the COSASS project, in which I will work during my doctorate. In this project, intelligent agents will use the resources of IoT artifacts, which will be subjected to extreme conditions, exposed to the elements, and prone to connectivity issues due to mountainous terrain and wildlife.

The inclusion of artifacts in FIVE requires these artifacts to have the ability to send and receive commands from the server. Similarly, we must be able to represent them in the IVE, and they must be able to store and display the information of the resource they provide. Additionally, they should also have a maximum interaction radius, because in the real world they may have antennas that limit the influence of communications with other agents.

Modeling artifacts in FIVE will allow us to conduct tests where these simulated artifacts will suddenly lose connection and stop providing service to the agents. Being able to represent these dynamic scenarios in a simulated three-dimensional environment before deploying the project in the real world is a very useful contribution that will save

researchers time and effort, in addition to reducing the economic cost of conducting tests in real environments.

3.1.5. Graphic User Aids

Graphic aids are always present to improve user interaction with complex systems, so it is interesting to add two features to facilitate our interaction with FIVE: rays that show the communication radius between agents and a new overhead camera that allows us to observe the environment from an elevated position quickly.

The representation of the agents' action radius also allows us to integrate a new feature into the FIVE framework, which is a neighbour list for each agent. This neighbour list would include the neighbours that are within our communication radius and would allow the study of dynamic graphs, where the topology would change based on the position of the agents.

3.2 Improvements to the ACoL Framework

The ACoL framework is a tool that allows for experiments with SPADE intelligent agents, providing a means to apply the Asynchronous Consensus-based Learning algorithm. Its execution is straightforward and it features a web page as a graphical interface to configure and launch the SPADE agents. Furthermore, we can integrate the ACoaL algorithm —proposed in this academic work— for use in a testing phase of the COSASS project, prior to conducting tests in the three-dimensional environment of FIVE.

However, this framework has limitations and therefore, we can add functionalities to improve its performance. One drawback of this tool is that experiments with agents that share the architecture of a small neural network (on the order of 25 500 parameters) work correctly, but when dealing with a larger network (for example, 98 500 parameters), it stops working and it is not possible to complete the execution phase of the algorithm. After a thorough analysis of the framework's code and its libraries, we have detected that this is due to the maximum accepted size of the XMPP messages. The tool encodes the weights of the neural networks in `base64` and sends them to a random neighbour agent using SPADE's XMPP messaging service, which in turn uses the XMPP messaging service of `aioxmpp` [4]. The `aioxmpp` library limits us to a maximum message size of $1024 \times 256 = 262\,144$ characters. For this reason, another very interesting improvement to the ACoL framework is to implement a messaging system capable of overcoming this limitation while maintaining compatibility with SPADE agents.

There are other minor improvement options that we can implement in the ACoL framework to make it more useful to researchers. Currently, the ACoL framework has two neural network models that we can select and two datasets: a Multi-Layer Perceptron (MLP), a Convolutional Neural Network (CNN), the MNIST dataset [17], and the Fashion-MNIST dataset [18]. However, only the MLP network with the MNIST dataset is functional, due to the message size limitation we have discussed. For this reason, the design, construction, and integration of a new CNN for recognizing disease patterns in citrus fruits is of interest, thus providing utility to the COSASS project.

On the other hand, we can add a new dataset that allows us to conduct tests with the ACoaL algorithm. Currently, in the COSASS project, we do not have a good dataset with which to conduct tests for detecting diseases in citrus fruits, so it is of interest to construct a new dataset with similar semantic information, to have the software prepared when

[4] https://pypi.org/project/aioxmpp/

the citrus dataset is available and at that moment it is not a limitation to conduct tests between the ACoL and ACoaL algorithms.

Finally, we can improve other minor details such as the configuration of algorithm's hyperparameters in the `Config.py` configuration file, or implementing a log file export system, useful for running many tests and quickly retrieving the results. As well as a mechanism to adjust the training and test samples that the different agents will receive. In this way, we can purposefully unbalance the classes and simulate a coalition of agents that traverse an orchard with one type of fruit trees and another coalition of agents that traverse another type of fruit trees.

3.3 Asynchronous Consensus-based with Coalitions Learning algorithm

The Asynchronous CoL algorithm combines the advantages of a decentralized and asynchronous version of Federated Learning, as we explained in section 2.9. This algorithm is designed to improve the efficiency of machine learning processes by distributing the training process across multiple clients and combining the models without the need for a central server. In FL algorithm each client trains a model on a dataset, and sends its model to the central server upon completion of a round of training. However, the central server in this setup can constitute a single point of failure as well as a possible bottleneck. To address this issue, the ACoL algorithm eliminates the need for a central server and allows the agents to communicate directly with each other.

The key specificity of the ACoL algorithm is that it removes the idle time of all clients, and therefore, the model training is performed quicker. In the ACoL setting, a round of training can be described as follows:

1. All clients start training their local models, using their local training data.

2. As soon as a client is done training, they send their model weights and biases to one random neighbour in the graph, and this neighbour responds to them with its current weights and biases.

3. The client then checks if they had received any messages during their training phase, and if it is the case, they apply model aggregation for every received message.

4. The client then starts the training phase again.

This asynchronous implementation enhances the solution as if one device is down, or takes too long, the other devices can still communicate and perform model aggregation. This shows that ACoL can present significant advantages over synchronous algorithms. Additionally, there is the possibility that the system is composed of agents that share common characteristics, such as the semantic similarity of the data on which they are training the model, or the position of the agents, which can affect their communicative capacity and the topology of the communication graph.

For this reason, the intuitive idea arises to form groups of neighbouring agents that communicate more among themselves than with agents outside their group. As we have discussed in section 2.6, communication between sets of agents is defined by the topology and modelled with an undirected graph $G = (A, E)$, where the nodes are the agents of the set A and the edges E are formed by pairs (a_i, a_j), denoting that agent

a_i is connected with agent a_j. Therefore, we denote the neighborhood of agent a_i with $N_i = a_j \in A : (a_i, a_j) \in E$.

These groups of agents facilitate the efficient use of resources. By working together, agents can pool their resources and capabilities, thereby achieving more than what would be possible if they were working individually. This is particularly important in scenarios where resources may be limited or costly. Moreover, coalitions can form dynamically, that is, changing the number of agents belonging to each coalition over time. They can also be static, if the groups are defined from the beginning and remain the same over time.

In this academic work, we propose the definition and implementation of the Asynchronous Consensus-based with Coalitions Learning (ACoaL) algorithm shown in Algorithm 3.1. This algorithm is based on the fact that all agents share the same neural network architecture to be able to apply the consensus algorithm [13] [19] on the weight matrix W_i. The parameters it takes as input are: e, the number of training epochs of the neural network before sending the weights; k, the number of matrices that will be agreed upon; c, the number of iterations of the consensus algorithm to be applied; p, the probability of sending the message to an active agent of our same coalition. This definition is an adaptation of the ACoL algorithm 2.2 where a random neighbour $n \in N_i$ is not chosen with a uniform probability distribution, but the probability of choosing it is based on whether the agent n belongs or does not belong to our coalition.

Algorithm 3.1 $ACoaL_{a_i}(e, k, c, p)$ - Asynchronous Consensus-based with Coalitions Learning algorithm for agent a_i

```
1:  while !doomsday do
2:    for t ← 1, e do
3:      W_i ← Train(f)
4:    end for
5:    for t ← 1, c do
6:      X_j = PROBABILITY_BASED_SELECT(N_i, p)
7:      for j ← 1, k do
8:        X_i^t ← W_j
9:        X_i^{t+1} ← (1 − ε)X_i^t + εX_j^t
10:       W_j ← X_i^{t+1}
11:     end for
12:   end for
13: end while
```

We will implement the ACoaL algorithm in section 4.3 and conduct a series of experiments in section 5.2.1 with the aim of drawing conclusions. Coalitions will be defined statically and we will study the influence of establishing groups based on the semantics of the data they process and based on the geographical position of the agents.

3.4 Geographical Threshold Graph - CoL

GTG-CoL, standing for Geographical Threshold Graph - Consensus-based Learning, is a novel approach to distributed learning that is specifically designed for Wireless Adhoc Networks (WANET). WANETs are formed by a set of mobile units, such as agents, equipped with a wireless connection antenna, allowing them to communicate in a decen-

tralized manner. These agents may have a short communication range due to antenna capabilities and the need for battery-saving.

In GTG-CoL, the agents communicate with others within the range of their wireless connection. The potential connections they can make as the agents move define a Geographical Threshold Graph (GTG), according to the maximal range given by the agents' antennas. This is particularly relevant in scenarios such as a fruit orchard with autonomous tractors, which is presented in section 5.3.

The GTG-CoL algorithm is based on the Co-Learning algorithm explained in section 2.8, so each agent trains a neural network independently during a certain number of epochs. The training results are a set of matrices, for each of which the agent performs several iterations of the consensus algorithm. This consensus is calculated with the models shared by the agents within their communication range.

This proposal, developed in collaboration with Dr. M. Rebollo, Dr. J. A. Rincón, Dr. L. Hernández, and Dr. C. Carrascosa, is particularly useful, both in this academic work and in the COSASS project. We will study the consensus speed of the agents depending on the communication graph, simulating WANETs using FIVE, whose topology will change over time. The topology of the communication graph will provide a dynamic formation of coalitions among agents, which will be of interest to study. However, for consensus to be achieved, we need the graph to eventually be a strong-connected component [13], so in the design phase we will construct the experiments to ensure this property is met at least at one point in time.

Finally, we will perform three types of tests that follow the theme of the COSASS project and integrate perfectly into FIVE, where we will study the network's efficiency and fault tolerance when an agent suddenly loses the ability to communicate with the rest, changing the coalition's structure. The tests we will complete are:

1. An orange orchard with tractor intelligent agents traversing each row of trees in parallel until reaching the final end.
2. Two orange orchards with agents moving perpendicularly.
3. We will expand the second test, adding an additional agent that performs random movements.
4. An additional experiment will be made to study how one tractor agent starting in one group of agents can help to reach the consensus by increasing its velocity and joining the other group of agents, using the FIVE framework.

CHAPTER 4
Solution design

4.1 Improvements to the FIVE framework

4.1.1. Agent Behavior Templates and Algorithm Plug-in System

Agent behaviours templates

SPADE intelligent agents running on the same FIVE client previously shared a single behaviour template, where the unique characteristics of each agent were programmed within this template. For instance, if a federated learning-based algorithm was used, the agents acting as servers and clients would be distinguished in the same code. Alternatively, two FIVE clients could be run, one for the agents acting as clients and another for the server. The process of programming agent behaviours is more comfortable, faster, and organized with the new agent behaviour template system.

The new template system allows the user to specify which behaviour template each agent will use, so in the same FIVE client, all the agents of the simulation can be configured independently. To build this system, the Python library `importlib` was used, which allows importing the code of a module by providing the file path, as shown in Figure 4.1.

```
1 class_name = "EntityShell"
2 behaviour_class_path = f"behaviours.{self.
      __behaviour_path}"
3 behaviour_class = getattr(importlib.
      import_module(behaviour_class_path),
      class_name)
```

Figure 4.1: Code that allows dynamically loading behaviour files.

To specify the individualized behaviour of the roles of the agents, we must follow these three simple steps:

1. We create as many behaviour files —with a descriptive name— in the `behaviours` folder as we need different roles for our agents. The `behaviours` folder is located within the FIVE client (`src/agents/Agents`).

2. We program the code of the agent behaviours within the files created in the previous step, using the `EntityShell` template class and overloading the methods of the agent's Finite State Machine (FSM).

3. We specify the behaviour of each agent in the configuration file `configuration.json`. The `behaviour` property of this JSON format file indicates the name of the file containing the definition of the `EntityShell` class with the behaviour of our agent.

To illustrate the simplicity of this new behaviour system, we will perform an example of this process. We will create a behaviour that consists of using the `change_color` command to change the agent's color as it transits the states of the FSM, as shown in Figure 4.2.

(a) Agent in initial state.

(b) Agent in perception state.

(c) Agent in cognition state.

(d) Agent in action state.

Figure 4.2: An agent going through the four states of the FSM associated with its behaviour and changing color after reaching each one.

First, we create a file with the name we want to give to the behaviour, for example, `color_behaviour.py` and include it in the behaviour container folder `behaviours`. Next, we write the behaviour code in the `EntityShell` class, overwriting the FSM methods. As in this case, we only change the agent's color, we call the `change_color` method of the agents themselves in each of the states, as shown in Figure 4.3.

```
1 await agent.change_color(1, 0, 0, 0.8)
2 time.sleep(4)
```

Figure 4.3: Code that runs during the agent's perception state in Figure 4.2. The first line of code tints the agent red, and the last line of code performs a wait of four seconds.

Finally, we set the `behaviour` property of the `configuration.json` file, as shown in Figure 4.4. This configuration allows FIVE to know which file to import with Python's `importlib` library, for each of the smart agents that will be created during execution.

```
{
  "name": "agent1",
  "at": "gtirouter.dsic.upv.es",
  "password": "xmppserver",
  "folderCapacitySize": 80,
  "imageFolderName": "captures",
  "enableAgentCollision": true,
  "prefabName": "Tractor",
  "position": "Spawner 1",
  "algorithms": [],
  "behaviour": "color_behaviour"
},
```

Figure 4.4: Extract from the modified `configuration.json` file.

Plug-in algorithm system

Intelligent agents are key components of various algorithms, such as FLaMAS, CoL, ACoL, etc. Therefore, there is an opportunity to improve the FIVE framework if the process of programming agent behaviours is streamlined, providing the user with the general aspects of the different algorithms and focusing the user's effort on programming only the specific aspects needed in their implementation.

The improvement has been built by adding a new option to the agent configuration file `configuration.json`, called `algorithms`. This option allows indicating a list of the algorithms that will be included in the agent's behaviour, for example, `"algorithms": ["flamas"]`. This system is designed so that more advanced users can define the base behaviour of the algorithms that the intelligent agents will perform, and the FIVE user only has to import these algorithms in the configuration file and complete the specific details in the agent behaviour code.

If we want to define the FLaMAS algorithm to be included within this system, we must establish a connection between the list of algorithms and the base code implementation of FLaMAS in FIVE. This connection is made in the `entity.py` class of the agents, and consists of a guard that checks if the user has included our algorithm within the agent's algorithm list. Then, the algorithm's behaviour is programmed in code files that we later import into `entity.py`. In the case of FLaMAS, it consists of an FSM with four states, so it can be implemented as an FSM behaviour in SPADE and included in the set of behaviours of the intelligent agent, as shown in Figure 4.5.

```
if "flamas" in self.__algorithms:
    # AGENT PRESENCE HANDLER
    flamas_presence_handler = StateFlamasPresenceHandler(behaviour_class)
    self.add_behaviour(flamas_presence_handler)

    fsm_behaviour = AgentBehaviour()

    # STATES
    fsm_behaviour.add_state(name=STATE_FLAMAS_SETUP, state=StateFlamasSetup(
        behaviour_class), initial=True)
    fsm_behaviour.add_state(name=STATE_FLAMAS_RECEIVE, state=StateFlamasReceive
        (behaviour_class))
    fsm_behaviour.add_state(name=STATE_FLAMAS_TRAIN, state=StateFlamasTrain(
        behaviour_class))
    fsm_behaviour.add_state(name=STATE_FLAMAS_SEND, state=StateFlamasSend(
        behaviour_class))

    # TRANSITIONS
    fsm_behaviour.add_transition(source=STATE_FLAMAS_SETUP, dest=
        STATE_FLAMAS_RECEIVE)
    fsm_behaviour.add_transition(source=STATE_FLAMAS_SETUP, dest=
        STATE_FLAMAS_TRAIN)
    fsm_behaviour.add_transition(source=STATE_FLAMAS_RECEIVE, dest=
        STATE_FLAMAS_TRAIN)
    fsm_behaviour.add_transition(source=STATE_FLAMAS_RECEIVE, dest=
        STATE_FLAMAS_RECEIVE)
    fsm_behaviour.add_transition(source=STATE_FLAMAS_TRAIN, dest=
        STATE_FLAMAS_SEND)
    fsm_behaviour.add_transition(source=STATE_FLAMAS_SEND, dest=
        STATE_FLAMAS_RECEIVE)

    flamas_weights_template = Template()
    flamas_weights_template.set_metadata("flamas", "weights")
    self.add_behaviour(fsm_behaviour, flamas_weights_template)
```

Figure 4.5: Extract of the code that runs during the agent's construction, and includes the states of the FLaMAS algorithm within the agent's FSM behaviour.

Using this system, the generic code of the algorithm (such as the control of sending the neural network weights between clients and server) is performed transparently for the user, and the FIVE user only needs to complete the specific details of the algorithm in the `EntityShell` file, saving a significant amount of effort and work.

4.1.2. Optimizing FIVE Networking System while Improving Capabilities

The network communication with the agents has been restructured to optimize message exchange and simultaneously gain new features, such as transforming the FIVE server into an agent capable of subscribing to the presence of other agents and vice versa.

To achieve this, the previous network socket system has been replaced, where the FIVE server managed the command socket and the image socket of each of the agents that connected to it, as shown in Figure 3.1. Note that in this initial approach, the number of active network sockets scales linearly with the number of intelligent agents that connect to the FIVE server, leaving room for improvement in managing network communications.

The new version of the FIVE server connects directly with the XMPP server used by the agents to communicate with each other. This not only solves the scaling problem but also transforms the FIVE server into a fully modular entity, decoupled from the FIVE

framework, allowing the FIVE server to be used with any other program that models intelligent agents. The new network topology can be seen in Figure 3.2.

Unity does not provide any mechanism for communication with XMPP servers, but as the programming language used is C#, there is a free package called S22.Xmpp [1] that allows the use of XMPP communication mechanisms. For this reason, it was decided to integrate this package into the FIVE framework and restructure the code to create a messaging system compatible with FIVE's SPADE agents.

This new communication system compatible with XMPP requires replacing the classes that managed TCP communication with a system of callbacks that trigger events and manage a thread-safe message queue to transport information between Unity threads. The class that performs these actions is called `XmppCommunicationManager`. When this class is instantiated, it uses the information provided by the user to attempt a connection with the XMPP server. This connection attempt is made in the same way as other SPADE agents do, by sending the node name and password, or by sending the node name and attempting In-Band registration, where the XMPP server allows access to a node that is not registered in the system. Figure 4.6 shows an excerpt of the code where this connection attempt is made.

```
1  private void AttemptRegistrationInBand() {
2      if (!xmppClient.Connected)
3          ConnectXmppClient();
4      try {
5          xmppClient.Register(
               RegisterInBandCallback);
6          xmppClient.Authenticate(xmppNode,
               xmppPass);
7      } catch (XmppErrorException ex) {
8          Debug.LogError(ex.Error.Text);
9          throw ex;
10     }
11 }
```

Figure 4.6: Extract of the FIVE code that attempts to register a node using the callback pattern.

The SPADE library includes metadata—transparent to the user—within its messages and for this reason, plain XMPP messages do not reach the SPADE intelligent agents. The solution to this problem is to create a class that generates an abstraction layer for the low-level construction of XMPP messages, and for the FIVE server to use it to communicate with the SPADE agents. In this way, when the FIVE server needs to transmit information to an agent, it uses the `XMPPCommunicator` class and embeds the necessary metadata so that the SPADE template system accepts the messages, instead of discarding them.

[1] https://www.nuget.org/packages/S22.Xmpp

Figure 4.7: FIVE configuration menu.

Finally, an adaptation of the FIVE configuration menu has been made to adjust the new features associated with XMPP communication. As shown in Figure 4.7, it also allows for the selection of viewing resolution (which works on 4K screens) and the selection of TLS message encryption, to provide greater privacy and security in the communications of intelligent agents and the FIVE server.

4.1.3. Including artifacts into the IVE

Integrating artifacts into IVEs significantly improves the modeling of projects that include them, as they are resources that intelligent agents can utilize through interaction. The inclusion of these new elements needs adaptations to the FIVE server's code to distinguish between an artifact and an agent. Therefore, we will detail the changes made to the FIVE server.

There is a new command called `CreateArtifactCommand` used to instantiate artifacts. Thus, when we want to intantiate an artifact, we only need to send a command via the XMPP protocol to the FIVE server's Jabber ID (JID) with the instantiation data in JSON format, following the FIVE command standard. Figure 4.8 illustrates the path that commands take within the FIVE framework, which we will describe in detail. Therefore, the attributes of the command message (formatted in JSON) sent to the FIVE server, are the following:

- `"commandName": "createArtifact"`
- `"data"`: This list of strings corresponds to the following options:
  - First string: Name of the IoT artifact to be displayed in the IVE.
  - Second string: Artifact prefab name, which is the name of the type of artifact to be instantiated.
  - Third string: Artifact position. This can be a string containing the name of a spawner object in the IVE, or the position in JSON format:

* "x": Numerical value corresponding to the X-axis position.
* "y": Numerical value corresponding to the Y-axis position.
* "z": Numerical value corresponding to the Z-axis position.

Once the command is received by the FIVE server, the plain text in JSON format is converted to the `CommandInfo` class, which encapsulates this generic information. Then, the `CommandParser` class must be able to translate and interpret this encapsulated information. Therefore, the `CreateArtifact` method has been created, which, given an instance of the `CommandInfo` class, generates an instance of the `CreateArtifactCommand` class completed with the artifact's specific information.

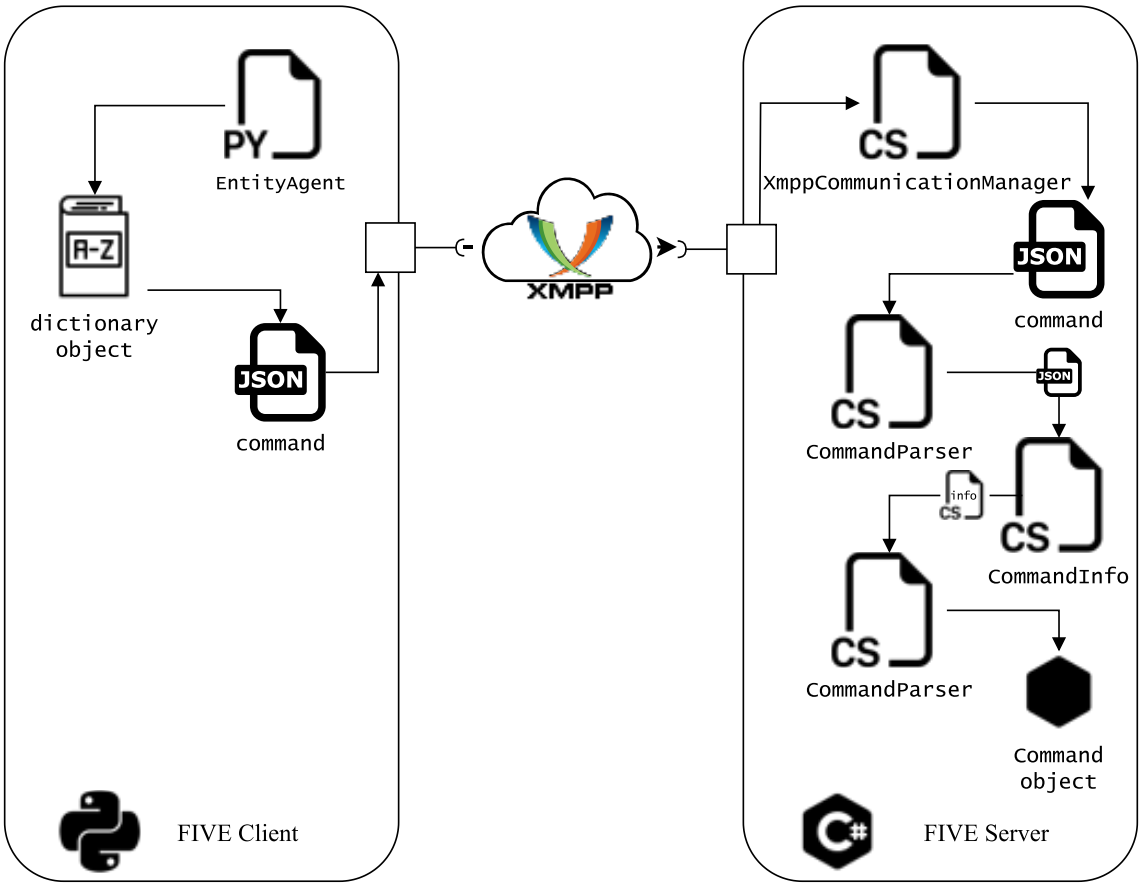


Figure 4.8: Scheme of the new FIVE command protocol.

The instance of the `CreateArtifactCommand` class allows the FIVE server to determine what type of artifact is to be instantiated and whether it will be instantiated using a position tag referencing the spawner's name, or alternatively, it will use an absolute position defining the exact coordinates. This specification is made in the `IsInstantiatedByCoordinates` method of the `CreateArtifactCommand` class.

Next, the command is stored in a thread-safe queue during the execution of the thread that manages the XMPP communication of the `XmppCommunicationManager` class. Later, the main Unity execution thread dequeues the commands from the thread-safe queue and executes them. The command to instantiate an artifact triggers the invocation of the `CreateArtifactEntity` method of the `XmppCommunicationManager` class, which creates a new entity of the new `Artifact` class that contains the code with the generic behavior of FIVE artifacts.

Finally, after the artifact is created, the `XmppCommunication` class configures its attributes using the information from the `CreateArtifactCommand` instance. At this point,

the actual artifact can interact with the avatar created by FIVE and modify the data linked to it. In the forthcoming section 5.1, we will explore how this process has been utilized in an IoT device that makes predictions of temperature and humidity based on past temperature, humidity, and pressure data using a regression model with neural networks.

This IoT device serves as a practical example of how artifacts can be integrated into the FIVE framework. The device's ability to predict environmental conditions based on historical data showcases the potential of artifacts in enhancing the capabilities of intelligent agents. The agents can use the predictions made by the IoT device to make more informed decisions, demonstrating the value of incorporating artifacts into IVEs.

4.1.4. Visual tools

In this section, we will explore the incorporation of two features aimed at enhancing user interaction with the FIVE framework. The first feature involves the use of rays that visually depict the communication radius between agents, offering a clear representation of their reach, as it is shown in Figure 4.9. This aids in understanding the scope and limitations of agent interactions within the IVE. The second feature introduces an overhead camera perspective that enables users to observe the environment from an elevated position, providing a broader view for better situational awareness. These new graphic features have been implemented with the collaboration of Dr. Luís Hernández López. Additionally, the integration of a neighbour list for each agent based on their communication radius allows for the analysis of dynamic graphs, where the network topology evolves based on the agents' positions. These graphic user aids contribute to an improved user experience and a more comprehensive understanding of the FIVE framework.

Figure 4.9: Side view of three tractor agents in a fruit orchard IVE with pink rays indicating their communication ability.

The rays that indicate the communication radius of the agents are managed by the `CheckNeighbors` method of the `Entity` class, which contains the logic of the three-dimensional model of the agent and performs the actions that the SPADE agents of the FIVE client send through the XMPP server. Agents check the distance to other agents by performing a vector calculation between their position and the position of the other agent. In this way, if the distance separating them is less than the maximum communication radius of the agent, then a new `GameObject` is created to draw the ray in the scene. This new object is given the `LineRenderer` component to be able to render the line in the IVE. Finally, the reference to this object is stored in a dictionary-type structure, which relates agents to their communication rays, so when the distance between our agent and the other agent exceeds the established threshold, we can refer to the dictionary and remove this object from the scene.

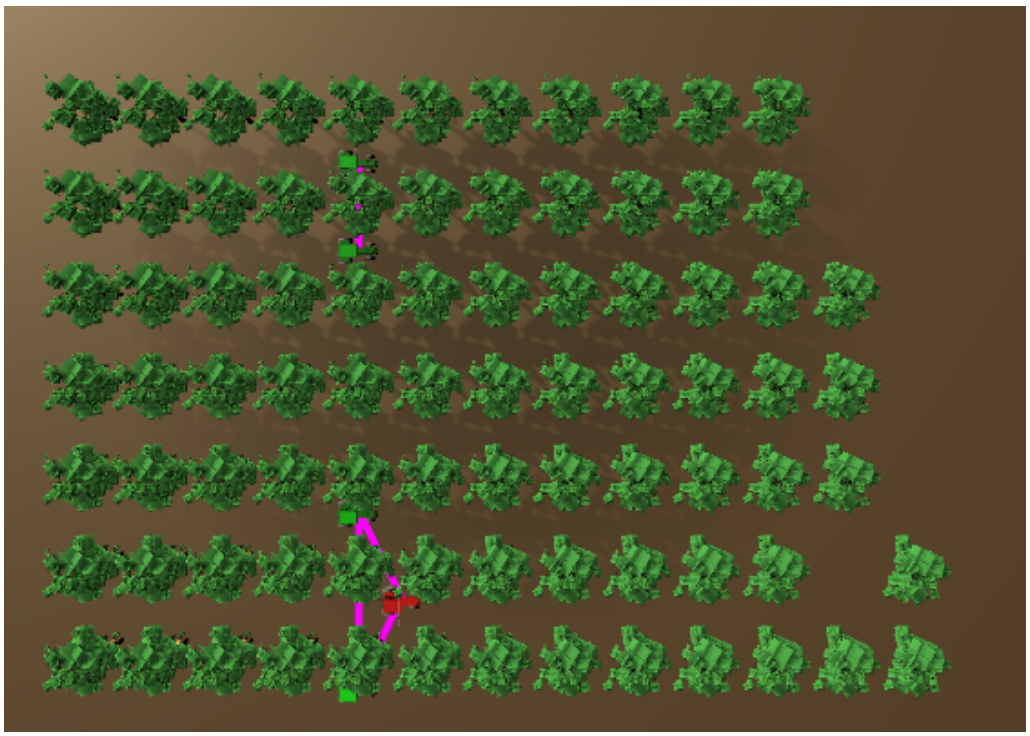


Figure 4.10: Overhead view of five tractor agents traversing a fruit orchard.

Finally, the overhead camera consists of an additional camera-type object, which has been added to the scene to provide an aerial view. As can be seen in Figure 4.10, this view helps the user to see all the agents participating in the IVE quickly. To activate this overhead camera, we must press the `T` key and we can quickly return to the main camera with the `M` key for free movement through the scene.

4.1.5. Conclusion

The enhancements made to the FIVE framework primarily boost the following aspects: improved network communication, which optimizes message exchange while reducing the computational load on the FIVE server; transforming the FIVE server into an agent that can subscribe to the presence of other agents, and vice versa, providing better integration with SPADE; enhancing the comfort of agent creation, providing a system of algorithm templates and individualized behaviours for each agent; and extending visual aids for the user, such as the integration of a top camera and the drawing of lines between agents to denote their communication radius.

To achieve these goals, modifications have been made to the communication channel between the FIVE server and the agents. Instead of creating a command socket and an image socket for each of the agents, the FIVE server now has the ability to connect with intelligent agents through the XMPP protocol. This change not only frees up the server's management of communications with the agents, but also unifies the command and image sockets by using XMPP message metadata to differentiate them while being sent over the same network socket.

We can also add that a complete decoupling has been carried out between the FIVE server and the SPADE agents. This advantage allows the FIVE server to connect not only with SPADE agents, but with any type of intelligent agent framework, greatly expanding the potential use of the FIVE framework by the scientific community.

Finally, the concept of an artifact has been added, allowing them to be incorporated into the virtual environment so that agents can make use of the resources they provide. In section 5.1, we will see the implementation of an IoT artifact in real-world equipped with a neural network that performs a regression model to predict the humidity and temperature of the next hour, given a history of 24 previous readings (taken every hour) of humidity, temperature, and pressure. It is important to note that the goal of this proof of concept is not to obtain an optimized neural network for the task, but to demonstrate the developed framework's ability to integrate these real-world IoT devices with useful resources for the agents.

4.2 Improvements to the ACoL Framework

This section focuses on the ACoL framework, which serves as a tool for conducting experiments with SPADE intelligent agents and applying the Asynchronous Consensus-based Learning algorithm. The main goal of improving the ACoL framework is to incorporate the Asynchronous Consensus-based with Coalitions Learning (ACoaL) algorithm and utilize it in the COSASS project. The framework features a web page as a graphical interface for configuring and launching SPADE agents. However, the current version of the ACoL framework has limitations that hinder its performance in certain aspects. One major drawback is the inability to execute experiments with large neural network architectures due to the maximum accepted size of XMPP messages. This limitation arises from the encoding and transmission of neural network weights, which exceed the message size limit set by the `aioxmpp` library. To overcome this constraint, an improvement to the framework is proposed by implementing a messaging system capable of surpassing this limitation while maintaining compatibility with SPADE agents. Additionally, there are other minor enhancements that can be made, such as the inclusion of a new convolutional neural network model for recognizing disease patterns in citrus fruits and the integration of a new dataset for testing the ACoaL algorithm. Further improvements involve configuring algorithm hyperparameters, implementing a log file export system for result retrieval, and adjusting training and test samples for different agents to simulate specific scenarios. These enhancements aim to make the ACoL framework more versatile and useful for researchers, particularly in the context of the COSASS project, where the integration of the ACoaL algorithm is of primary interest.

4.2.1. Solution to XMPP message size restriction

The solution to the size limitation imposed by the `aioxmpp` library on which SPADE is based, is founded on creating a system capable of breaking down the body of the messages to a maximum configurable size. The new system for sending and receiving mes-

sages must be able to convert a SPADE message that exceeds the maximum XMPP message size into several smaller messages that do not exceed this imposed limitation and at the same time must be able to reconstruct the original message in the correct order and transparently for the user. It is very important that this maximum size is easily configurable by the user, because the size that SPADE reserves to include the metadata of the messages can vary, depending on the use we are giving to the ACoL framework. In this way, if we exceed the maximum size of the XMPP message, we only need to reduce the maximum size we allow in the body of the SPADE message and the new system will correctly manage the sending and receiving of messages.

The new messaging management system developed consists of a new class called `MultipartHandler`. This class performs two phases: a phase of message sending management and a phase of receiving them. Both phases work in parallel and we will first explain the operation of the message sending phase.

The sending of messages requires the partitioning of them into smaller fragments. For this reason, we must control the labeling of them, to ensure that the reconstruction is done in the correct order. First, we check if the length of the body of the SPADE message exceeds the desired maximum size. This is done by the `generate_multipart_messages` method that receives the content of the message, the desired maximum size of the fragments and the original SPADE message to automatically generate a template with the same metadata that we are using in our algorithm. This method, part of whose implementation is shown in Figure 4.11, adds the `multipart#` header followed by an identifying code of the messages, which also provides information about the total number of fragments required to reconstruct the message.

```
def generate_multipart_messages(self, content: str, max_size: int, message_base
    : Message) -> list[Message]:
    multiparts = self.multipart_content(content, max_size)
    if multiparts is not None:
        multiparts_messages = []
        for multipart in multiparts:
            message = copy.deepcopy(message_base)
            message.body = multipart
            multiparts_messages.append(message)
        return multiparts_messages
    return None

def multipart_content(self, content: str, max_size: int) -> list[str]:
    if len(content) > max_size:
        multiparts = self.divide_content(content, max_size)
        return [f"multipart#{i + 1}/{len(multiparts)}|" + part.strip() for i,
            part in enumerate(multiparts)]
    return None
```

Figure 4.11: Extract of the class `MultipartHandler` that manages the generation of messages.

Once we have all the fragments generated with their corresponding label, the `SEND` state of the FSM of the agent sends them to the target agent. Each state that requires sending or receiving messages, contains an instance of the `MultipartHandler` class to manage both the sending and the possible reception of messages. This class manages the phase of receiving messages with an attribute of the dictionary type, where the key is the node of the agent that has sent the messages and the value is an ordered list that contains the body of the messages without the header that this class adds to identify them. In this way, we can control the storage and position of them. For this process to be as transparent as possible for the user, there is the `rebuild_multipart` method shown in 4.12. Each

time a message is received by a state of the FSM of a SPADE agent, this state uses the `rebuild_multipart` method to automatically rebuild the complete message, in case the message provided as a parameter is a fragment and all the fragments have been received. This behaviour is designed so that only when a message is complete, the method returns it and thus we can control that we have all the information. If the message is not complete, it stores it in the dictionary attribute, in the correct order for a later reconstruction when all its parts are available.

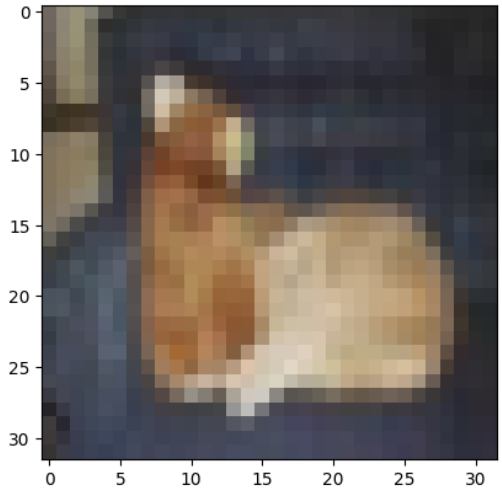
```
def rebuild_multipart(self, message: Message) -> Message:
    if self.is_multipart(message):
        sender = str(message.sender)
        multipart_meta = message.body.split('|')[0]
        multipart_meta_parts = multipart_meta.split('#')[1]
        part_number = int(multipart_meta_parts.split("/")[0])
        total_parts = int(multipart_meta_parts.split("/")[1])
        if not sender in self.multipart_message_storage.keys():
            self.multipart_message_storage[sender] = [None] * total_parts
        self.multipart_message_storage[sender][part_number - 1] = message.body[
            len(multipart_meta + '|'):]
        if self.is_multipart_complete(message):
            message.body = self.rebuild_multipart_content(self.
                multipart_message_storage[sender])
            del self.multipart_message_storage[sender]
            return message
    return None
```

Figure 4.12: Extract of the class `MultipartHandler` that manages the reception of messages.

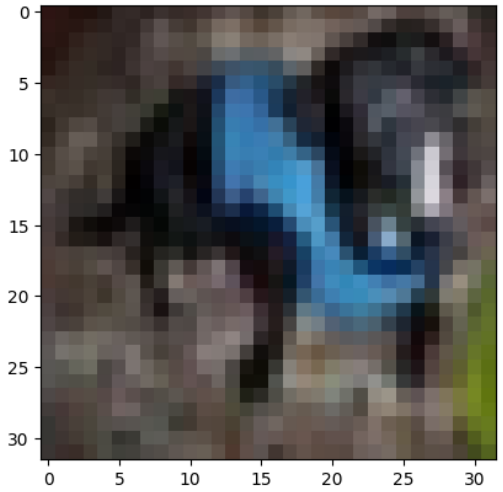
Finally, this design has been tested for the sending and receiving of weights of large neural networks and works perfectly even if the message division generates 100 fragmented messages at once. In section 5.2.1 it will be used to send weights of an MLP neural network between several SPADE intelligent agents.

4.2.2. Integration of a new dataset

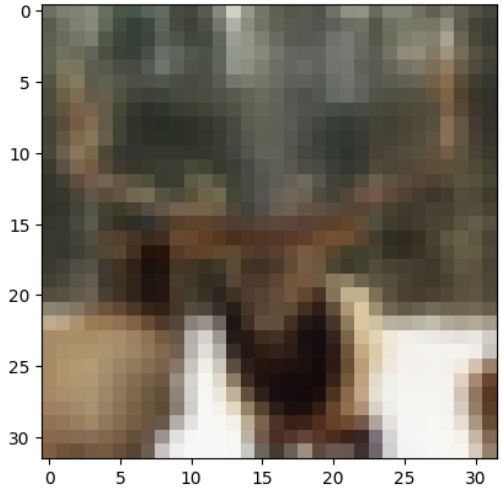
The ideal situation would be to have a balanced and complete dataset with different images of healthy fruits and also affected by different diseases to train a neural network with generalization capacity and with the aim of using it in the COSASS project, but currently we do not have such a thing. To make a very simplified approximation of this scenario, we are going to resort to creating a new dataset from a subset of the Canadian Institute for Advanced Researchers (CIFAR) dataset, in its reduced version CIFAR10 [20]. This dataset consists of 60 000 (32×32) colour images in 10 classes, with 6 000 images per class. There are 50 000 training images and 10 000 test images.



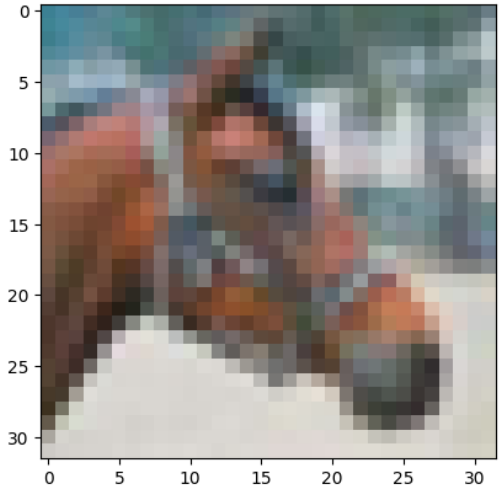
(a) Image of a cat in CIFAR4.



(b) Image of a dog in CIFAR4.



(c) Image of a deer in CIFAR4.



(d) Image of a horse in CIFAR4.

Figure 4.13: Images of CIFAR4 dataset.

We will create a subset of CIFAR10, which we will call CIFAR4, based only on the images of four classes: cats, dogs, deer, and horses. The example of some images is shown in Figure 4.13 and the reason for this selection is to be able to carry out tests with images that share a similar semantics. In this case, they are all mammalian animals, where we can say that cats and dogs share a certain similarity, and deer and horses also share it. This will allow us to carry out experiments provisionally until we have a complete citrus dataset, and it will also provide more diversity of data sets to the ACoL framework.

4.2.3. Design of a new convolutional neural network

The reason why we design and implement a new convolutional neural network is to add more diversity of types of networks to the ACoL framework and also to carry out tests on the new CIFAR4 dataset, which we have described in section 4.2.2. The new neural network will be implemented in two versions to work both in TensorFlow [2] and in PyTorch [3], and the experiments can be consulted in the Jupyter Notebook [4] `cifar4.ipynb`.

[2] https://www.tensorflow.org/
[3] https://pytorch.org/
[4] https://jupyter.org/

The architecture of this neural network is represented in Figure 4.14 and consists of the concatenation of two convolutional layers with one max-pooling, twice. Then, a flatten is performed and it is connected to two dense layers of 512 and 4 neurons. All activation functions are ReLU [21] except for the last layer, which is a softmax.

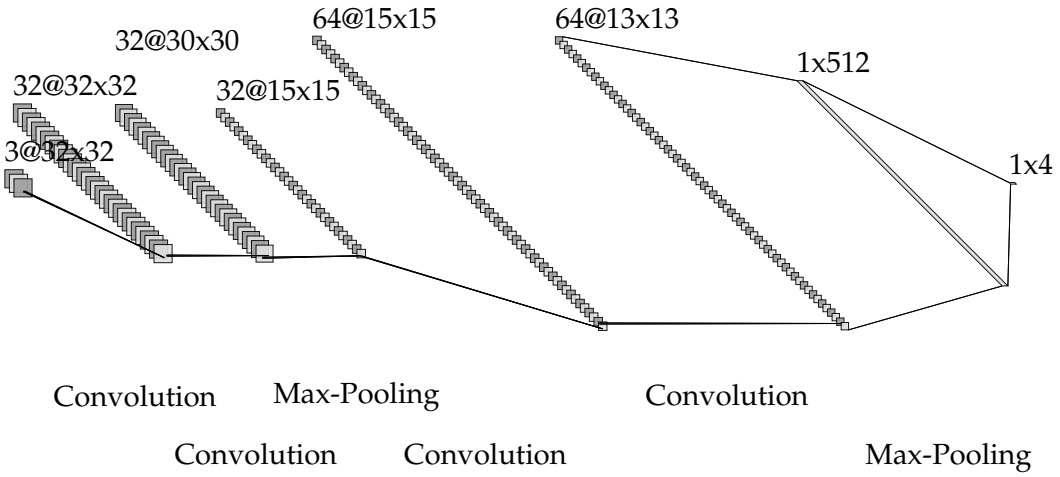


Figure 4.14: Scheme of the CIFAR4 CNN.

The ReLU function introduces non-linearity into the neural network model. Non-linearity is essential because it enables the network to learn and represent complex, non-linear relationships between input variables and the output. The ReLU function is a piecewise linear function that simply outputs the input value if it is positive and sets it to zero otherwise. This non-linear behavior allows the network to model more complex and expressive relationships in the data.

On the other hand, the softmax function normalizes the output values across multiple classes, ensuring that they represent probabilities. It maps the raw output values to a probability distribution, where each class probability is between 0 and 1 and the sum of all probabilities is 1. This property makes softmax suitable for classification tasks, as the output probabilities can be interpreted as the likelihood of the input belonging to each class.

Also, to improve the generalization capacity of the network, two dropouts with a probability of 0.1 have been added after performing max-pooling and a dropout with a probability of 0.25 before the last layer. Max-pooling helps in creating translation invariance, which means the network becomes insensitive to small translations or shifts in the input data. By taking the maximum value within a local neighborhood, max-pooling retains the most salient features while discarding less relevant or noisy information. This property makes the network more robust to variations in the input data, enhancing its ability to generalize well to unseen examples.

On the other hand, dropout acts as a form of regularization by introducing noise and reducing the interdependencies among neurons. By randomly dropping out neurons, dropout prevents the network from relying too heavily on specific neurons or co-adaptations among them. This encourages the network to learn more robust and generalizable representations of the data, reducing overfitting and improving performance on unseen examples. In total, the network is composed of 1 247 780 parameters, which occupy a total space of 4.76 MB.

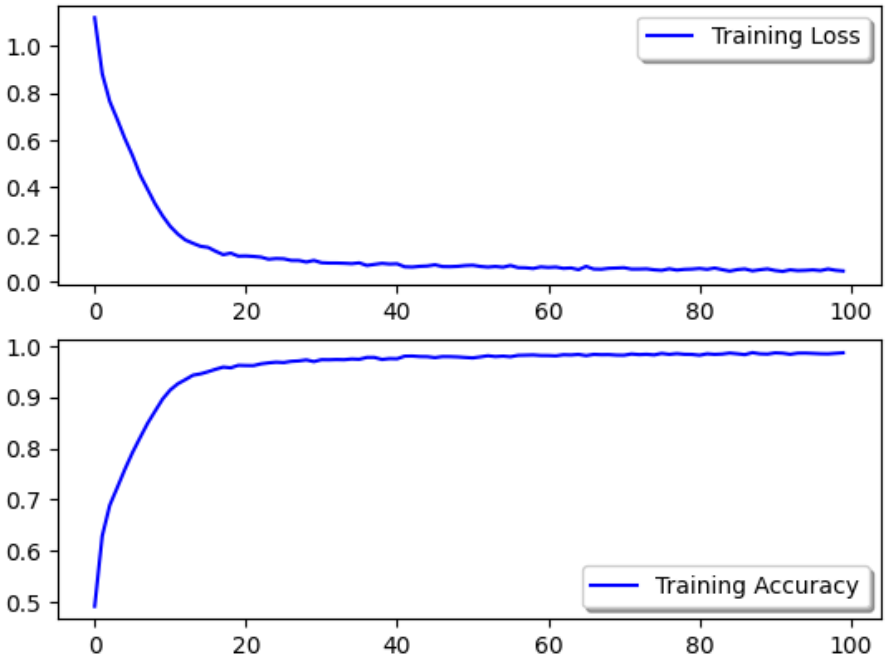


Figure 4.15: Accuracy and loss during training phase.

Once the architecture is implemented, we can train the network with a set of 20 000 training images among the four classes and we will test it with a set of 4 000 test images. Before training the network, we will apply a data preprocessing technique that consists of normalizing the values within the range $[0, 1]$. As the data are images, we know that the values can only reach 255, therefore, we can divide the data by 255 to fix the values.

The one-hot encoding technique is also interesting, which transforms numerical class labels into a binary vector of zeros and a single one in the class that corresponds to the numerical label. This conversion enables the network to treat each category as a distinct feature with a specific weight associated with it. By assigning separate weights to different categories, the network can capture the importance of each category in predicting the target variable accurately.

Finally, we set the loss function as categorical crossentropy, as it is a categorical classification problem, and the optimizer will be `AdamW`. The hyperparameters to start the training will be: learning rate of 0.001, batch size of 32 samples and we will train it for 100 epochs.

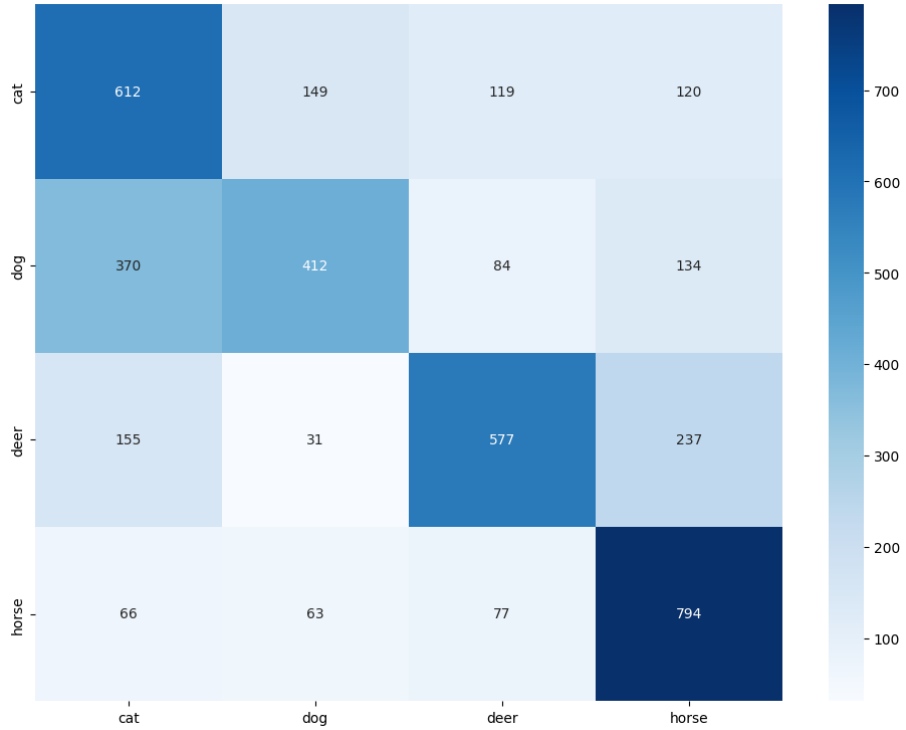


Figure 4.16: Test confusion matrix.

The training results are shown in Figure 4.15 and the test results are shown in Figure 4.16. We can observe that the network has successfully learned the training data, as the loss function is very low and the accuracy is close to 1. On the other hand, the confusion matrix (where the real labels are the rows and the predictions are the columns) that has been generated with the test images is very interesting, because although the result has been positive and has classified most of the samples correctly, it can be seen that the network has greater difficulties in differentiating cats and dogs than horses and deers.

4.2.4. Improvement of the ACoL framework configuration

The ACoL framework is much more complete if we increase the amount of options that the user has to be able to configure it. For this reason, we have added new functionalities that may be interesting to add to the existing ones. Now there is the option to configure that the log files are automatically exported in two different ways: adding them all together to a specified path or automatically creating folders to organize each of the experiments carried out separately. The functionality of being able to change the name of the folders of each of the logs from the configuration file itself has also been added.

On the other hand, the default behaviour of this framework when it detected the checkpoint of a trained network model was to load it. An option has been added to disable this behaviour in case you prefer to start the execution with the untrained weights. The option to configure the maximum message size has also been added, which as we have seen in detail in section 4.2.1 is essential for agents to be able to send the weights of their models. The maximum time that reception states wait for messages has been exposed in the configuration file, because depending on the characteristics of the network it may be convenient to modify them.

The incorporation of the ACoaL algorithm in the ACoL framework also includes the configuration of four parameters. On the one hand, there is the probability with which the agents send messages to the members of their coalition, which we have adapted to be

able to use it as a toggle between the ACoL algorithm, if this variable takes the value -1 and ACoaL if it is positive, in which case it would indicate the probability of emitting a message to another agent of the same coalition as the agent that is in execution. To know which agents are members of which coalitions, a list of lists of coalitions has been defined, where the elements denote the name of the agents that belong to the same coalition. There is also the maximum number of iterations or total epochs that the agents are going to perform. Finally, we have a dictionary that allows us to adjust the samples that the agents of the different coalitions will receive. This is especially useful for unbalancing the samples and that the agents of a coalition only receive images of cats and dogs, and those of another coalition only receive images of horses and deer, in the example of the CIFAR4 dataset. This last feature will be widely used in the COSASS project, for example, so that the agents of a coalition receive images of different types of orange diseases and the agents of another coalition receive images of different types of lemon diseases, with the purpose of conducting experiments.

4.3 ACoaL algorithm implementation into ACoL Framework

The implementation of ACoaL into the ACoL framework primarily involves developing the system for coalition assignment and neighbor selection for message sending. The `FLAgent` class contains the logic of the FL agent, which is why the code that assigns the coalition to which the agent belongs has been included in its constructor. An excerpt of this code is shown in Figure 4.17, and as we can see, it uses the configuration from the `Config.py` file to go through the static coalitions defined by the user, thereby assigning the correct coalition neighbours to the agents.

```
coalition_index = -1
for i, coalition in enumerate(Config.coalitions):
    if self.name in coalition:
        self.coalition = [a for a in coalition if a != self.name]
        coalition_index = i
        break
```

Figure 4.17: Extract of the code `FLAgent` that manages the coalition set.

Another key aspect in ACoaL is the selection of the neighbour to whom we are going to send the message with the weights, shown in Figure 4.18. The logic of the method must check within the list of active neighbours (`active_agents`), which agents belong to our coalition and which other agents are available. This is because even though the probability of sending a message to the agents in our coalition (`coalition_probability`) is higher than sending it to another agent not in our coalition, there may be no active agent from our coalition at the time of sending, in which case, we would send the message to an agent not in our coalition. On the other hand, if there are active agents, both from our coalition and the rest, we will use the randomly generated probability to check whether we should send it to an agent in our coalition or to another agent.

```
def pick_agent(self, coalition: list[str], active_agents: list[str],
    coalition_probability: float) -> str:
    random_prob = random.random()
    coalition_agents = [a for a in active_agents if a in coalition]
    other_agents = [a for a in active_agents if a not in coalition]
    if coalition_agents and (random_prob < coalition_probability or len(
        other_agents) == 0):
        return random.choice(coalition_agents)
    return random.choice(other_agents)
```

Figure 4.18: Extract of the code `SendState` that manages the agent selection.

Finally, by configuring the probability of choosing an agent from our coalition in the `Config.py` file, we can activate or deactivate the ACoaL algorithm of the ACoL framework. If we select a coalition probability less than 0 (such as -1), the ACoaL algorithm would be deactivated, and we would always choose a random active agent. Otherwise, the coalition probability would indicate the probability of sending a message to an active member of our coalition.

4.4 GTG-CoL

The study of the GTG-CoL system and the execution of tests, described and proposed in section 3.4, depends on their design. Therefore, it is crucial to properly design both the orange orchards and the communication radius of the WANETs.

The orchards are created using the IVE designer integrated into the FIVE framework. This system can convert a plain text file into an environment with dynamic textures, loaded at runtime. Therefore, we will write the desired shape of the orchard using ASCII characters in the `map.txt` file, and the FIVE system will take care of converting it into its representation in three-dimensional models. The result of applying this process to obtain a virtual representation of a real orchard is shown in Figure 4.19.

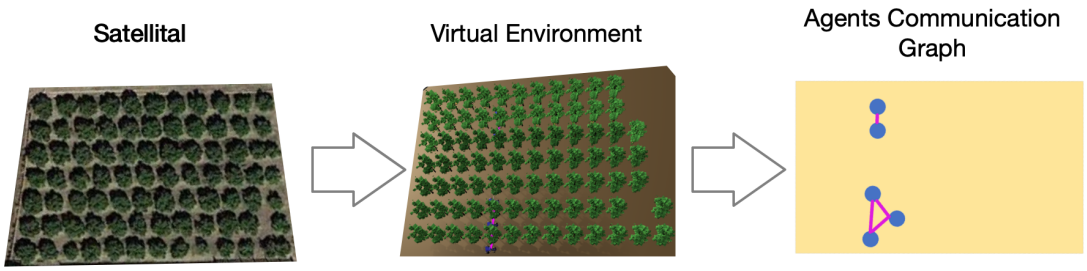


Figure 4.19: Real orchard into FIVE with tractor intelligent agents.

Next, we set the communication radius of the agents to the desired value depending on the experiment we are testing, using the default tractor agent model included in FIVE. This value has been chosen because it is sufficient for the agents to have at least one neighbour. Thanks to the ease of conducting tests in FIVE and the implementation of the visual aids we discussed in section 4.1.4, the radius has been easily validated before starting the testing phase.

CHAPTER 5
Case of Study

5.1 IoT temperature and humidity artifact into FIVE IVE

5.1.1. Introduction

This case of study is an intersection of IoT devices, MAS artifacts, FIVE and neural networks for weather forecasting. The central component of the project is an IoT artifact based on an ESP32 M5Stack Core2, shown in Figure 5.1, it is a versatile development kit with a rich set of features suitable for IoT applications. The ESP32 M5Stack Core2 is equipped with WiFi connectivity, facilitating data exchange with various online resources and services.

The IoT artifact will predict the temperature and humidity of the next hour based on a window of 24 hourly measurements of temperature, humidity, and pressure. To achieve this, we train a Multilayer Perceptron (MLP) neural network using historical weather data. Out of these, the network is embedded onto the ESP32 M5Stack Core2 artifact. This is accomplished using the `tensorflowlite_esp32` library, which enables the deployment of TensorFlow Lite models on ESP32 microcontrollers.

Figure 5.1: ESP32 M5stack Core2 IoT device.

The IoT artifact communicates with a Django endpoint via an HTTP client. This endpoint serves two main functions. Firstly, it makes requests to weather APIs to gather real-time weather data. Secondly, it serves normalized temperature, humidity, and pressure data, which the IoT artifact uses to make its predictions.

In addition to the HTTP client, the IoT artifact also uses custom XMPP templates to communicate with the FIVE server. This allows the artifact to send its updated predictions to the server, contributing to a comprehensive and continually updated weather model resource for the IVE agents.

The project also leverages the FreeRTOS library, a real-time operating system for microcontrollers, to manage various tasks that need to be executed over time. These tasks include sending updated predictions to the FIVE server, requesting normalized weather data from the Django endpoint, and consulting a Network Time Protocol (NTP) server to maintain accurate timekeeping.

5.1.2. Dataset

The "Historical Hourly Weather Data" [1] dataset is a comprehensive collection of weather data that spans over a period of approximately five years. It provides hourly measurements of various weather attributes such as temperature, humidity, and air pressure. This dataset is not only valuable for meteorological studies but also serves as a rich resource for a wide range of interdisciplinary research areas.

The dataset was originally collected with the intention of demonstrating basic signal processing concepts, such as filtering, Fourier transform, auto-correlation, and cross-correlation. The weather data, with its periodic temporal structure, offers an excellent platform to illustrate these concepts. However, the potential applications of this dataset extend far beyond this original purpose.

In our project, we will be using this dataset to train a neural network for predicting temperature and humidity. The theme of this project consists of an IoT artifact based on an ESP32 M5Stack Core2 that predicts the temperature and humidity of the next hour based on a window of 24 hourly measurements of temperature, humidity, and pressure. The neural network will be trained as a regression model using the "Historical Hourly Weather Data" dataset.

The dataset was acquired using the Weather API on the OpenWeatherMap website and is available under the ODbL License. It covers 30 cities in the US and Canada, as well as 6 cities in Israel. Each attribute (temperature, humidity, air pressure, etc.) is stored in a separate file, with rows representing the time axis and columns representing different cities. This organization of data facilitates easy use and analysis. Additionally, for each city, the dataset also provides information about the country, latitude, and longitude.

The "Historical Hourly Weather Data" dataset is a valuable resource for our project. It will enable us to train a robust neural network model that can accurately predict temperature and humidity based on past weather data, thereby improving the capabilities of our IoT artifact.

5.1.3. Neural Network architecture

We have constructed a Multilayer Perceptron (MLP) neural network to create a regression model. Given the measurements of temperature, humidity, and pressure from a 24-hour window, this model can provide a prediction for the temperature and humidity of the next hour. This network is built using the Keras framework and its architecture consists of:

[1] https://www.kaggle.com/datasets/selfishgene/historical-hourly-weather-data

1. An initial flatten layer that reduces the input tensor (with dimensions 24×3) to a vector of 72 elements.

2. A dense layer of 128 neurons with a ReLU activation function, contributing 9 344 parameters.

3. Dropout with a probability of 0.25 for better model generalization.

4. A dense layer of 32 neurons with a ReLU activation function, contributing 4 128 parameters.

    - An output neuron with a linear activation function for temperature.
    - An output neuron with a ReLU activation function for humidity.

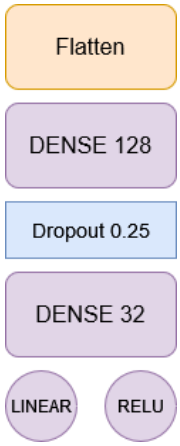


Figure 5.2: Top-down scheme of the Neural Network architecture.

The architecture of this network consists of a total of 13 538 parameters. The mean squared error (MSE) is used as the loss function on the validation set. The optimizer used is `AdamW` from the `TensorFlow` library, along with two callbacks: `ReduceLROnPlateau`, which automatically reduces the learning rate (LR) if there is no improvement over a certain period, and `ModelCheckpoint`, which saves the best model obtained. The hyperparameters used in `ReduceLROnPlateau` are: LR reduction factor of 0.1, patience of 30 units, and minimum LR of 0.000001.

The dataset has been processed to use only the temperature, humidity, and pressure data from Los Angeles, resulting in 39 969 samples of 24-hour measurement windows in training and 4 442 samples in the test set, i.e., the dataset is split into 90% train and 10% test. The hyperparameters used to initialize the neural network are 0.001 for LR, 128 for batch size, and 200 for epochs. We have conducted a variety of tests by adjusting the values of the described hyperparameters, and this configuration has yielded the most favorable results.

Finally, we trained the neural network on Google Colab [2] and achieve a Mean Squared Error (MSE) of 0.00075 on the test set. Following this, we utilize the `TFLiteConverter` module to convert the decimal values of our neural network model weights into eight-bit integers. This process optimizes the model's space, enabling us to embed it within the IoT device.

[2] https://colab.research.google.com

5.1.4. Django API endpoint and web services

The Django framework of the Python programming language has been used to build a web server that provides an API endpoint. This web server's purpose is to offer the 24 readings of temperature, humidity, and pressure to the IoT device in JSON format. These readings are served already normalized, so the IoT device does not have to perform the computational effort.

To allow the Django server to offer these data, we first need to create a space to store them. We will use an `SQLite` [3] database along with Django's model integration to create a `WeatherMeasure` class whose attributes will correspond to the measurements of temperature, humidity, and pressure, as well as a timestamp to indicate the temporal moment at which it was measured. Once we have defined this class, we can use the `WeatherMeasure.objects.get_or_create` method to create a reading in the database, or get it if it was already registered.

The Django server populates the database through requests to the Visual Crossing [4] API. First, a query is made to the URL of weather.visualcrossing.com that retrieves the hourly weather data of Valencia. Next, the `populate_data` method we have implemented takes care of processing and traversing the JSON readings to extract useful information about temperature, humidity, and pressure, and stores it in the database.

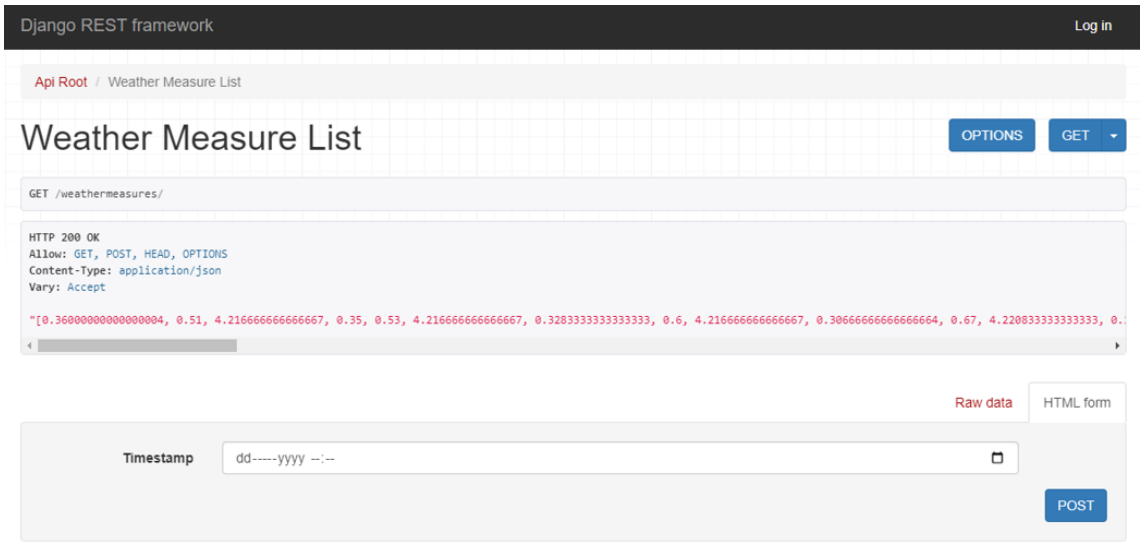


Figure 5.3: Django API endpoint serving weather data.

Finally, the endpoint is created and configured in Django so that the stored data can be served in the correct format, i.e., normalized for the regression model contained in the IoT device, resulting in the website shown in Figure 5.3.

5.1.5. Integration in FIVE

FIVE is a framework that has been improved with the perspective of being used both in scientific research and to be useful in the COSASS project. For this reason, the new update requires that the IoT device has the ability to communicate with the XMPP servers to be able to use the presence and message exchange for command sending.

[3] https://www.sqlite.org/index.html
[4] https://www.visualcrossing.com/

The ESP32 M5Stack Core2 device is equipped with WiFi connectivity, so we can make use of it using the `WiFi.h` library. An XMPP template system has been developed to provide the device with the ability to send commands to the FIVE server. This system developed in the Arduino IDE allows loading a basic XMPP template with the generic information that the FIVE framework needs and automatically completes the specific information of the IoT device, such as: the position where it will appear in the IVE, the type of IoT device (in this case it is a device for predicting temperature and humidity, and in the environment it appears as an icon formed by a thermometer and two drops of water) and the name that appears on the device's screen.

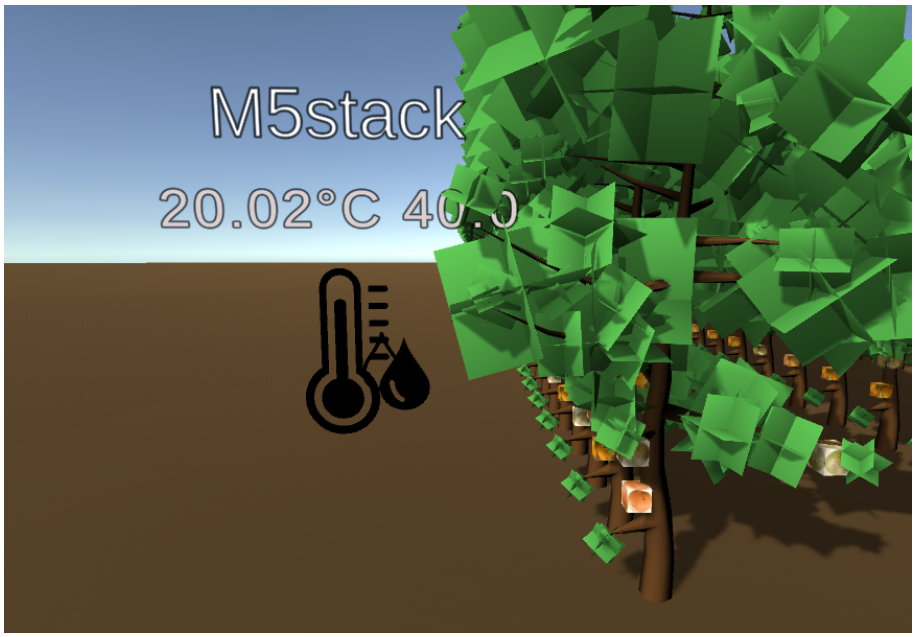


Figure 5.4: Aspect of the temperature and humidy IoT predictor device in the FIVE IVE.

When the IoT device starts, it first connects to the WiFi network and the XMPP server. Then, it configures the NTP server to have access to the current time. After that, it sends a command to create the artifact within the IVE to the FIVE server, using the XMPP template system described. Finally, using Free-RTOS, it executes a periodic task every minute, called `send_data_predict`.

The `send_data_predict` task consists of two phases. The first phase consists of obtaining the previous 24 climate readings using the API developed on the Django server. The second phase transforms the JSON data provided by the API into `float` type and then converts them back to `int8` to use them as input to the developed neural network and make predictions that will be subsequently sent to the FIVE server.

5.1.6. Free-RTOS

FreeRTOS is a real-time operating system (RTOS) designed for embedded systems, which is a crucial component in our IoT weather prediction project. It provides a robust and efficient environment for managing multiple tasks, which is essential for the complex operations of our IoT device.

In the context of our project, FreeRTOS allows the ESP32 M5 Stack Core2 to perform multiple tasks concurrently, such as collecting weather data, running the neural network model for prediction, communicating with the FIVE server via XMPP, and connecting

to the Django endpoint for getting the normalized data and synchronize time using the NTP server.

By efficiently managing these tasks, FreeRTOS ensures the real-time performance of our IoT device, enabling it to respond promptly to changes in weather conditions and to interact seamlessly with the FIVE IVE. This makes FreeRTOS a key enabler of our project's goal to integrate IoT capabilities into the IVE.

5.2 ACoaL performance study in MNIST and CIFAR4

In this section, we will conduct two experiments with two datasets: MNIST and the dataset we constructed in section 4.2.2 from a subset of CIFAR10 samples. The experiments were carried out in the facilities of the Department of Computer Systems and Computation (DSIC) [5] at the Polytechnic University of Valencia.

The ideal case would be to have access to a complete and balanced dataset with samples of healthy citrus fruits and with different diseases, which also share semantic information, such as oranges and lemons. Unfortunately, in the COSASS project, we do not have a dataset yet with these characteristics, and for this reason, we will conduct the experiments with the proposed datasets, which, despite not being fully suitable for this task, are well-known datasets and can serve as a baseline for future tests where a better dataset will be available to test the ACoaL algorithm.

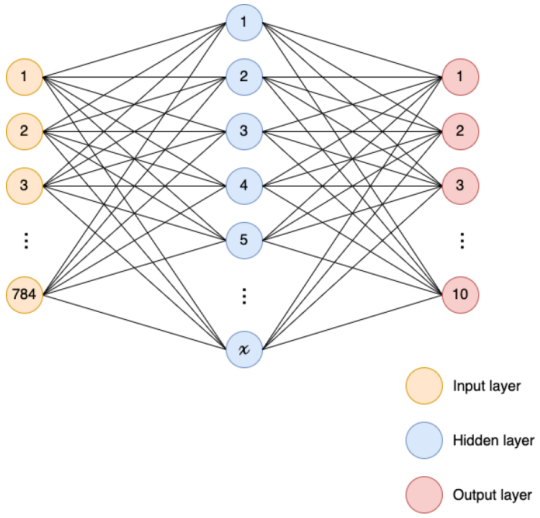


Figure 5.5: MLP neural network used in MNIST experiment.

In both cases, we will use an MLP neural network with one hidden layer, as described in Figure 5.5. On the other hand, in both experiments, the connection graph of the agents is complete; all agents have all other agents as neighbours at all times. After presenting each experiment, the results will be analyzed, and the conclusions obtained will be discussed.

Finally, it is worth noting that thanks to the extension of the ACoL framework that we carried out in section 4.2, we can experiment with the CIFAR4 dataset. The CIFAR4 dataset contains images with three channels (RGB) and of size 32×32, so the size of the neural network would make it unfeasible without the message partitioning system developed.

[5] http://www.upv.es/entidades/DSIC/index.html

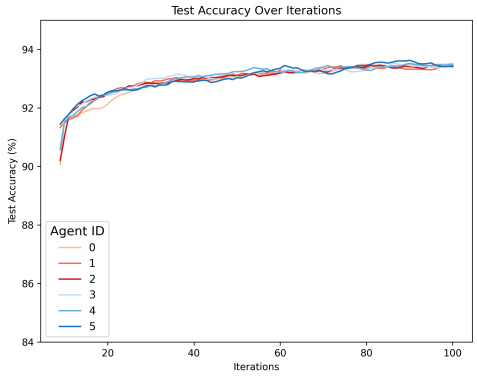
5.2.1. ACoaL using MNIST dataset

The objective of the following proof of concept is to show that coalitions are not always advantageous for any scenario. Still, they have greater potential when the individuals of the same coalition share a dataset with closer semantics than the rest of the groups. For this reason, the MNIST dataset [17] has been chosen, which consists of a set of handwritten digits with a training set of 60 000 examples, and a test set of 10 000 examples.

We will conduct four tests with the MNIST dataset identically distributed among all agents:

1. Test 1. Test the ACoL algorithm to obtain a baseline.
2. Test 2. Test the ACoaL algorithm with probability 0.6.
3. Test 3. Test the ACoaL algorithm with probability 0.8.
4. Test 4. Test the ACoaL algorithm with probability 0.9.

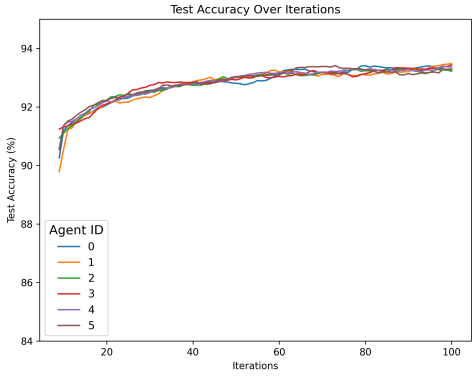
For this, six intelligent SPADE agents have been deployed using the improved ACoL framework detailed in section 4.2, and they have been configured from the `Config.py` file as explained in section 4.2.4. The configuration made consists of all agents performing 100 iterations of an epoch (100 epochs in total) and stopping their execution. Once the experiment is finished and its data has been processed for correct viewing, we can see the result of the accuracy on the test set in Figure 5.6.



(a) Accuracy in ACoaL algorithm ($p = 0.6$).

(b) Accuracy in ACoaL algorithm ($p = 0.8$).

(c) Accuracy in ACoaL algorithm ($p = 0.9$).



(d) Accuracy in ACoL algorithm.

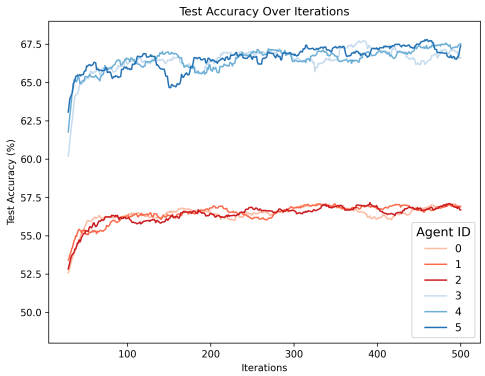
Figure 5.6: Accuracy on test using MNIST dataset.

We can appreciate that there is no significant difference when applying each of the tests, this is because the data used are not the most suitable to test the ACoaL algorithm. On the other hand, there is also no efficiency improvement in message sending when compared to the ACoL algorithm, because both algorithms choose a single neighbouring agent in the sending of weights. In the following experiment, we will have a dataset that has better semantics between classes and we will unbalance the data to study how it affects the performance of the algorithms.

5.2.2. ACoal using CIFAR4 dataset

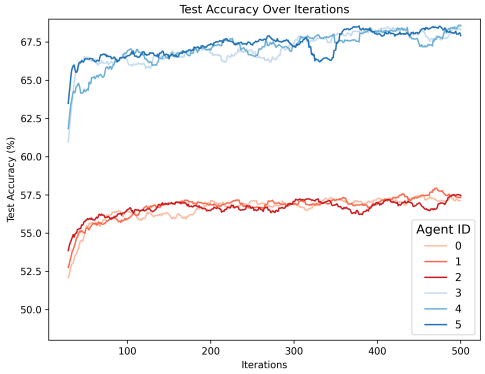
The CIFAR4 dataset is a reduced set of the CIFAR10 dataset, which we built in section 4.2.2. CIFAR4 has four classes, instead of ten, which are: cats, dogs, deer and horses. The following experiment consists of distributing the training samples among the agents to enhance the effect of the coalitions. Three agents (0, 1 and 2) will receive the samples associated with the classes of cats and dogs, both for train and test. On the other hand, the other three agents (3, 4 and 5) will receive the samples associated with deer and horses, both for train and test. The agents will be trained for 500 iterations of an epoch (500 epochs in total). Finally, the ACoaL algorithms will be tested with probabilities 0.6, 0.7 and 0.8 of sending a message to a neighbour in our coalition. Ultimately, we will test the ACoL algorithm.

The evolution of accuracy in the test dataset can be seen in Figure 5.7, where in this case there is a clear differentiation in the accuracy obtained by the agents of one group of samples and the other. In section 4.2.2 we saw the confusion matrix of the neural network we designed for CIFAR4, where it was appreciated that it was more difficult to distinguish between the classes of cats and dogs, than of horses and deer. This corresponds with the graphs obtained in this experiment, where agents 0, 1 and 2 obtain a worse accuracy than agents 3, 4 and 5.



(a) Accuracy in ACoaL algorithm ($p = 0.6$).

(b) Accuracy in ACoaL algorithm ($p = 0.8$).



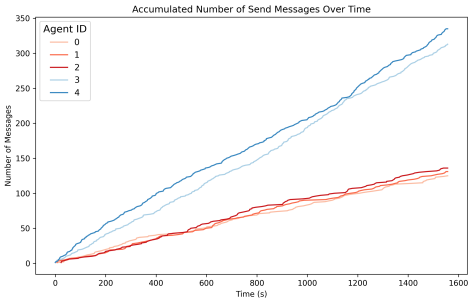
(c) Accuracy in ACoaL algorithm ($p = 0.9$).

(d) Accuracy in ACoL algorithm.

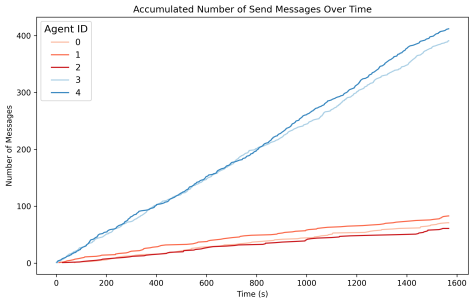
Figure 5.7: Accuracy on test using CIFAR4 dataset.

It is also observed how a higher probability of communication between the agents of the same coalition improves accuracy. This makes sense because we are validating the agents with sets of samples from their own training class. In a real experiment, we could have a set of intelligent agents that manage tractors and patrol an orange orchard, and there could be another adjacent lemon orchard with its corresponding agents as well. In this fictitious case, the agents would also have separate sets of samples, although they share the same neural network architecture. Intuitively it is interesting that this happens, because the agents patrolling the lemon orchard could learn to detect a disease that affects lemons and that can spread to oranges. In this way, they could transmit that knowledge to the agents patrolling the orange orchard (at a slower speed than their exchange of messages, due to the probability of sending a message to an agent who does not belong to our same coalition) before the lemon disease spreads to the orange orchard.

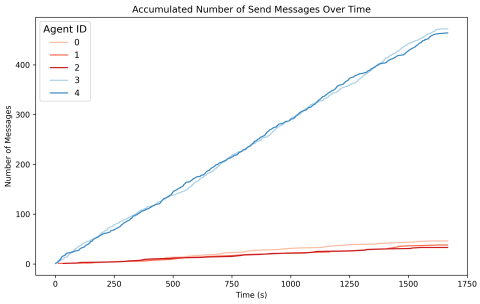
On the other hand, we can see the graph of messages sent from agent 5 to the other agents, in Figure 5.8. Agent 5 has been chosen randomly, because the result graphs of the other agents describe the same shape as those of agent 5. In these graphs, the influence of the probability of sending messages to members of our coalition is clearly appreciated, with the graph with $p = 0.9$ showing the greatest separability between the agents of the two coalitions. Finally, the graph of the ACoL algorithm does not discriminate against the agents and agent 5 has sent a similar number of messages to all the other agents.



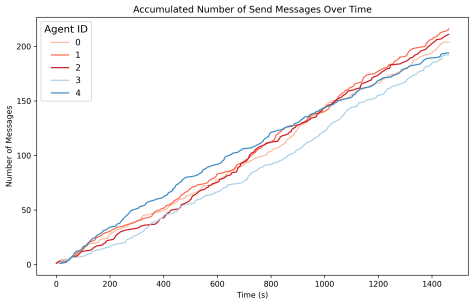
(a) Messages sent in ACoaL algorithm ($p = 0.6$).



(b) Messages sent in ACoaL algorithm ($p = 0.8$).



(c) Messages sent in ACoaL algorithm ($p = 0.9$).



(d) Messages sent in ACoL algorithm.

Figure 5.8: Messages sent by agent 5 to the other agents during the CIFAR4 dataset experiment.

5.3 GTG-CoL: A Simulation in an Orange Orchard

In this case study, our focus is on various scenarios where agents control tractors moving through a fruit orchard. The tractors' movements and their ability to communicate with each other are limited by their antenna range. As the tractors move, their set of communicable agents changes, creating a dynamic, switching communication network topology.

First, we will study the influence of tractor movement on the communication graph and therefore, on the dynamic coalitions formed by the agents. The aim of these tests is also to analyze the fault tolerance of random sudden disconnections and carefully selected disconnections. To do this, without loss of generality, we will project the space of the orange orchard into a two-dimensional space where we will carry out the three experiments described in section 3.4, analyze the results, and use the IVE built in section 4.4 to conduct an additional test and validate the consensus convergence.

Test 1. This test consists of a single orange orchard with tractor agents traversing each row of trees in parallel until they reach the final end. The agents have different speeds, which causes the restructuring of the communication graph topology at different moments in time, but the communication radius size is fixed so that they always have at least one neighbour. The result can be seen in Figure 5.9, where we can also appreciate the position of the agents and the communication graph they formed at times $t = 20$ and $t = 90$. The image on the right indicates the accumulated connections that have occurred between the agents during the test, with the line thickness between two nodes being greater when the agents representing these nodes have been connected more times during the simulation.

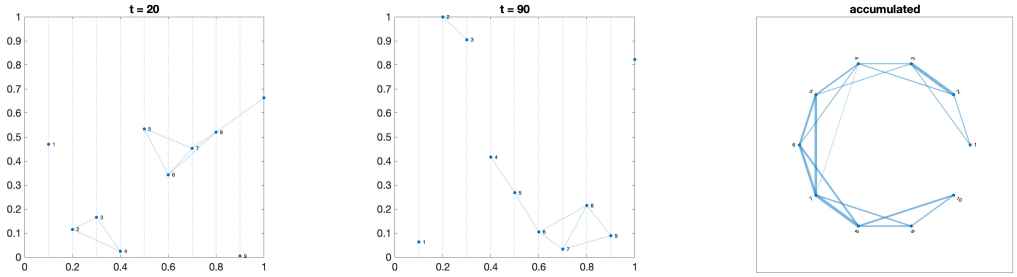


Figure 5.9: Network topology evolution as agents move in Test 1.

Test 2. This test consists of two orange orchards with tractor agents moving perpendicularly. The agents also have different speeds, and the communication radius is also fixed so that they always have at least one neighbour. The result is shown in Figure 5.10, where we can also appreciate the position of the agents and the communication graph they formed at times $t = 20$ and $t = 90$. Note that the horizontal and vertical lines in the first two subfigures represent the arrangement of the fruit trees. Therefore, the agents on the left make vertical movements, and those on the right make horizontal movements. Finally, the image on the right indicates the accumulated connections that have occurred between the agents during the simulation.

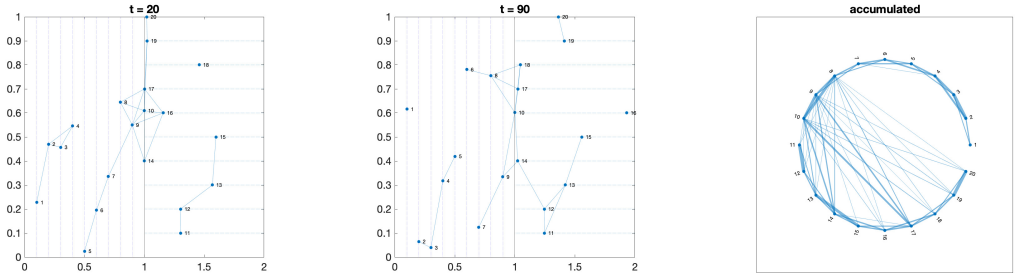


Figure 5.10: Network topology evolution as agents move in Test 2.

Test 3. This test involves adding to Test 2 an agent with no movement restrictions, specifically a flying drone with the ability to communicate with the rest of the agents. Furthermore, the behavior of this new agent consists of making random movements through the orange orchard. Figure 5.11 shows the result of this test, where the color red has been used to represent the new agent and its connections.

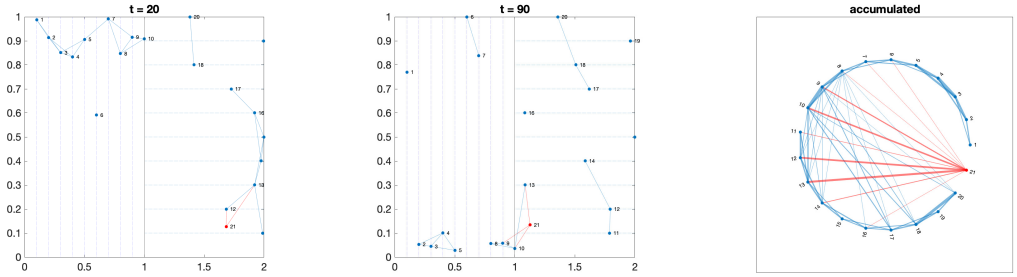


Figure 5.11: Network topology evolution as agents move in Test 3.

The analysis of the efficiency of the convergence process during the CoL algorithm requires comparing the average degree and the average shortest path of each aggregated network, in the three test scenarios. Networks with a higher degree need to exchange more information with neighbors, so the computational cost increases. On the other

hand, longer paths cause the time it takes for agents to agree on a value to be longer. The evolution of these measures as a function of the size of the agents' communication radius is shown in Figure 5.12, where we can see the stabilization effect caused by the agent with free random movement in the graph on the right.

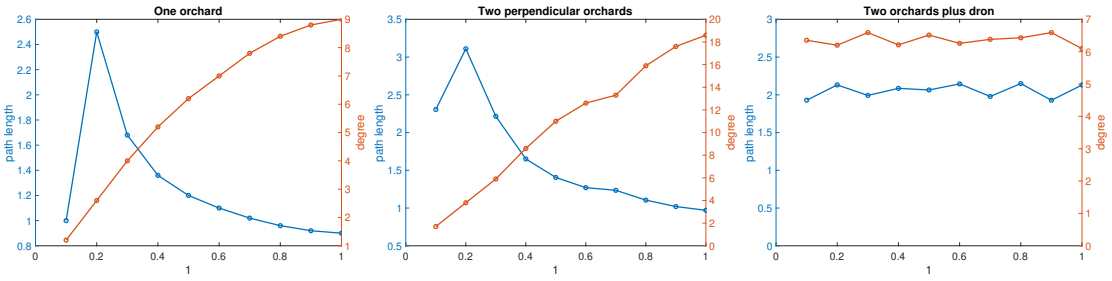


Figure 5.12: Evolution of the average shortest path length and degree.

The last analysis we will perform on these tests consists of studying the tolerance to random failures and carefully selected ones. We can see in Figure 5.13 how random failures do not significantly influence the behavior of the tests, this is a consequence of being a GTG, where spatial constraints avoid the existence of hubs. We can also see that the failures that have been deliberate, targeting the agents with the highest number of connections, degrade the network's efficiency more. This is because the agents targeted for removal are bridges between agent neighborhoods, and disconnecting them loses communication between the subgroups. Therefore, these patrolling agents are critical for the proper functioning of these systems.

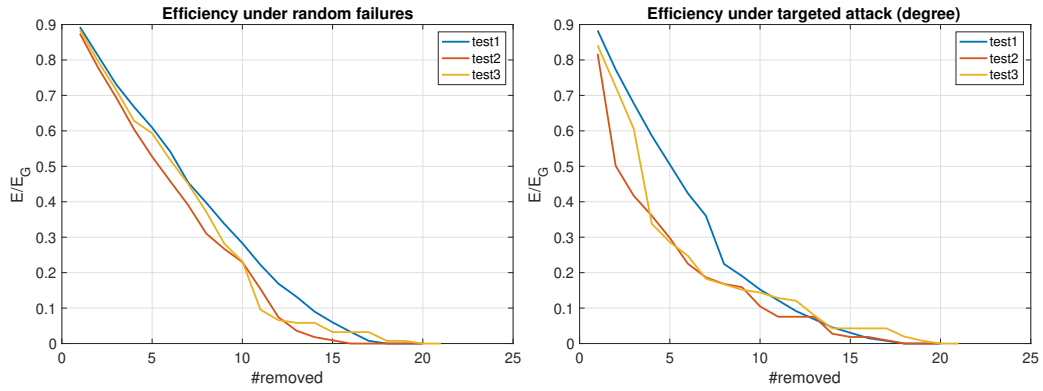


Figure 5.13: Evolution of the tests under random failures and deliberate agent removals.

Finally, we will conduct an experiment where the tractor agents will be set in the IVE that we have designed in section 4.4, so that they will form a group of two and another group of three agents, as Figure 5.14 shows. In the group of three agents, tractor number 2 (the one closest to the other group) will advance at double the speed of the rest of the agents. This increase in speed will create a strongly connected component when it advances to the next line of orange trees, and the information from the previously isolated agent groups will be agreed upon.

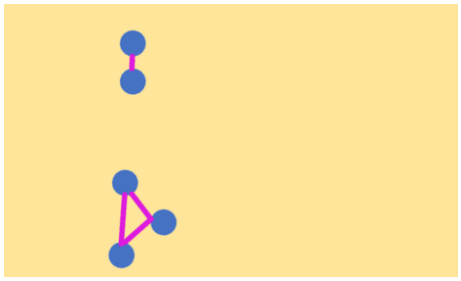


Figure 5.14: Agents' communication initial graph.

After running the experiment, we can appreciate the result in Figure 5.15. The graph on the left shows the connection graph at time $t = 160$, when the component is strongly connected. In the center, we can see the aggregated network, where a greater connection between the groups that have remained together since the beginning of the simulation is appreciated. Tractor agent number 2 acts as a link between the two subgroups and allows unifying the communication graph. The last graph shows the evolution of the consensus focused on a weight of the neural network of the CoL algorithm.

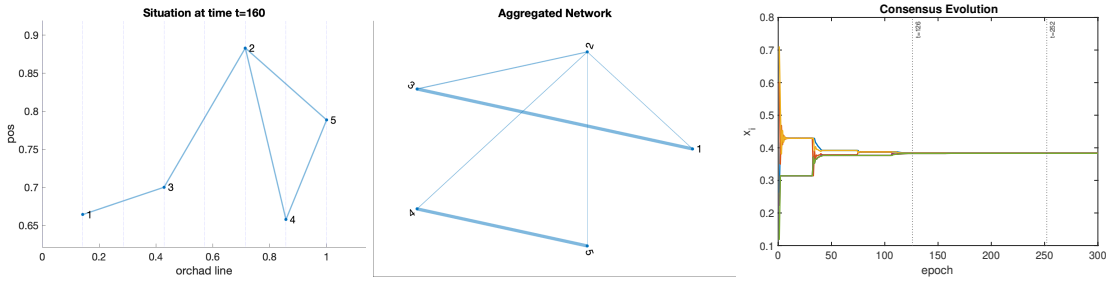


Figure 5.15: Result of the two initial subgroups of agents in an orange orchard.

As we can see, the agents have reached a consensus on the value and have stabilized after completing the union of the communication graph. The mean absolute error at the first vertical mark ($t = 126$) is MAE $= 0.0016$ and at the second mark ($t = 252$) is MAE $= 0.0000088$. Therefore, we can conclude that this is the necessary time for the neural network to be agreed upon.

CHAPTER 6
Conclusions

6.1 Synthesis

The expansion in the field of multi-agent systems calls for more efficient algorithms and new tools that streamline the research process. In this academic work, we have made several improvements in each of these aspects. We have designed and implemented the Asynchronous Consensus-based with Coalitions Learning algorithm, which allows the formation of coalitions among groups of intelligent agents applying decentralized and asynchronous federated learning. We have also improved the framework of the Asynchronous Co-Learning algorithm, enabling it to work with more complex neural networks, and endowed it with new functionalities, such as the implementation of ACoaL. This will help improve the efficiency of researchers dedicated to the study of multi-agent systems.

We have also significantly improved the capabilities of the Flexible Intelligent Virtual designer (FIVE) framework, integrating the concept of artifact into our simulations and providing the user with a new system of code templates for faster and more organized project programming. This has allowed us to test it with IoT devices that integrate artificial neural networks and study new algorithms like GTG-CoL, as well as contribute to scientific research with the article on this algorithm and the COSASS project, which is strongly linked to the deployment of agents and IoT artifacts in fruit orchards. On the other hand, the optimization of communication management when connecting to an XMPP server as another agent has also completely decoupled the abstraction layer that FIVE generates over the SPADE intelligent agents and the FIVE server itself that contains the IVE virtualization. For this reason, the FIVE server can be used with any program that has the capacity to communicate with an XMPP server, expanding the generalization capacity and versatility of the FIVE framework.

Finally, we can say that we have successfully achieved the main objective of this academic work, as well as all the secondary objectives we set at the beginning of the work.

6.2 Future works

The future development lines that arise from this project are mainly oriented towards the study of coalitions, the improvement of the exposed frameworks, and the creation of datasets to contribute to the COSASS project.

- Study dynamic semantic coalitions. Coalitions cease to be static and immovable, instead, intelligent agents can change coalitions at runtime and the algorithm must adapt.
- Creation of a dataset designed for the study of coalitions in federated learning-based algorithms.
- Add new types of sensors to the agents of the FIVE framework.
- Create a system for saving FIVE simulations, which allows for their viewing to be reproduced in the future.
- Study of optimization systems in message sending to achieve lighter and faster consensuses, such as the transformation of neural network weights into integers.

The source code of the project is located in two GitHub repositories. The repository of the FIVE framework and another repository with the rest of the source code implemented during this academic work:

- FIVE repository: https://github.com/FranEnguix/five
- General repository: https://github.com/FranEnguix/TFM_MIARFID

Bibliography

[1] M Rebollo, JA Rincon, L Hernández, F Enguix, and C Carrascosa. Gtg-col: A new decentralized federated learning based on consensus for dynamic networks. In *International Conference on Practical Applications of Agents and Multi-Agent Systems*, pages 284–295. Springer, 2023.

[2] Francisco Enguix Andrés. *Desarrollo de un generador de simulaciones en Unity 3D para sistemas multi-agente basados en SPADE*. PhD thesis, Universitat Politècnica de València, 2022.

[3] Michael Wooldridge and Nicholas R Jennings. Intelligent agents: Theory and practice. *The knowledge engineering review*, 10(2):115–152, 1995.

[4] Alessandro Ricci, Michele Piunti, and Mirko Viroli. Environment programming in multi-agent systems: an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2):158–192, Sep 2011.

[5] Michael Wooldridge. *An introduction to multiagent systems*. John wiley & sons, 2009.

[6] Michael Luck and Ruth Aylett. Applying artificial intelligence to virtual reality: Intelligent virtual environments. *Applied artificial intelligence*, 14(1):3–32, 2000.

[7] Talal Rahwan. *Algorithms for coalition formation in multi-agent systems*. PhD thesis, University of Southampton, 2007.

[8] Javier Palanca, Andrés Terrasa, Vicente Julian, and Carlos Carrascosa. SPADE 3: Supporting the new generation of multi-agent systems. *IEEE Access*, 8:182537–182549, 2020.

[9] Michael Bratman. *Intention, Plans, and Practical Reason*. Cambridge: Cambridge, MA: Harvard University Press, 1987.

[10] Unity automotive uses tech from gaming to aid automakers | digital trends. https://www.digitaltrends.com/cars/unity-automotive-virtual-reality-and-hmi/, 2018. Consultado en [10 Junio 2022].

[11] Soluciones | unity. https://unity.com/es/solutions. Consultado en [11 Junio 2022].

[12] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.

[13] Reza Olfati-Saber and Richard M Murray. Consensus problems in networks of agents with switching topology and time-delays. *IEEE Transactions on automatic control*, 49(9):1520–1533, 2004.

[14] Yanjiang Li and Chong Tan. A survey of the consensus for multi-agent systems. *Systems Science & Control Engineering*, 7(1):468–482, 2019.

[15] Stefano Savazzi, Monica Nicoli, and Vittorio Rampa. Federated learning with cooperating devices: A consensus approach for massive iot networks. *IEEE Internet of Things Journal*, 7(5):4641–4654, 2020.

[16] Jaime Rincon, Vicente Julian, and Carlos Carrascosa. Flamas: federated learning based on a spade mas. *Applied Sciences*, 12(7):3701, 2022.

[17] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[18] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017. cite arxiv:1708.07747Comment: Dataset is freely available at https://github.com/zalandoresearch/fashion-mnist Benchmark is available at http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/.

[19] 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, 2007.

[20] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[21] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.