Consensus-Based Learning for MAS: Definition, Implementation and Integration in IVEs

C. Carrascosa, F. Enguix, M. Rebollo, J. Rincon *

VRAIn - Valencian Research Institute for Artificial Intelligence, Universitat Politècnica de València, Valencia (Spain)

Received 23 February 2023 | Accepted 4 August 2023 | Published 24 August 2023

Abstract

One of the main advancements in distributed learning may be the idea behind Google's Federated Learning (FL) algorithm. It trains copies of artificial neural networks (ANN) in a distributed way and recombines the weights and biases obtained in a central server. Each unit maintains the privacy of the information since the training datasets are not shared. This idea perfectly fits a Multi-Agent System, where the units learning and sharing the model are agents. FL is a centralized approach, where a server is in charge of receiving, averaging and distributing back the models to the different units making the learning process. In this work, we propose a truly distributed learning process where all the agents have the same role in the system. We suggest using a consensus-based learning algorithm that we call Co-Learning. This process uses a consensus process to share the ANN models each agent learns using its private data and calculates the aggregated model. Co-Learning, as a consensus-based algorithm, calculates the average of the ANN models shared by the agents with their local neighbors. This iterative process converges to the averaged ANN model as a central server does. Apart from the definition of the Co-Learning algorithm, the paper presents its integration in SPADE agents, along with a framework called FIVE allowing to develop Intelligent Virtual Environments for SPADE agents. This framework has been used to test the execution of SPADE agents using Co-Learning algorithm in a simulation of an orange orchard field.

Keywords

Complex Networks, Distributed AI, Multiagent Systems, Neural Networks.

DOI: 10.9781/ijimai.2023.08.004

I. Introduction

This paper follows a research line related to multi-agent learning. So, it extends the work presented by Carrascosa et al. [1], where a new algorithm based on Federated Learning and Consensus in Multi-Agent Systems named CoL was presented. This extension focuses in how this kind of algorithms can be tested in execution in a close to real simulation using a new Intelligent Virtual Environment (IVE) generator.

Multi-agent Learning is currently a hot topic mixing machine learning with distributed systems. It can be found two main different kinds of such systems: the ones where the learning is a specific part of the system carried out by one (or a few) agents of the system (like in the work by Sánchez San Blas et al. [2]) where the deep learning is made by an agent in a complex system dedicated to the automatic detection of illegal swimming pools), and the ones where all agents make the same kind of deep learning process (that is, the learning also uses a distributed approach). In this last kind of system is where the proposed algorithm is classified. The proposed algorithm intends to get the most out of a distributed approach. It tries to mix the learned parameters in each agent with the parameters trained in its local neighbors without knowing the whole system. Moreover, this kind of learning algorithm preserves the privacy of the data used for the learning process by each agent in his local learning.

These features are, in some way, present in other approaches, mainly Federated Learning (FL). The FL algorithm was defined by Google [3]. The main idea behind this algorithm is to take advantage of distributed learning and maintain the privacy of the data used by each node in the learning process. The algorithm uses two different kinds of agents: server and client. The server defines the training model and sends it to all the clients. Then, clients train with their private data and send the model back to the server. Finally, the server aggregates all the models, for example, calculating an averaged one. This global model is sent back to execute the next training iteration. Kairouz et al. [4] analyze deeply the open challenges related to FL algorithms. It should be emphasized that the connection topology among the agents significantly influences the convergence rate in decentralized distributed learning processes. Nonetheless, the FL approach has noteworthy characteristics worth considering. Firstly, it maintains a distributed nature while operating with a centralized framework, implying that the system synchronizes and evolves based on the pace of the slowest agent. Furthermore, it lacks fault tolerance, rendering it vulnerable in scenarios where agents fail to respond or vanish, and it does not accommodate the incorporation of new agents during execution.

These features are of great importance when developing systems that must work in environments with a high probability of communication

* Corresponding author.

E-mail addresses: carrasco@dsic.upv.es (C. Carrascosa), fraenan@inf.upv.es (F. Enguix), mrebollo@dsic.upv.es (M. Rebollo), jrincon@dsic.upv.es (J. Rincon).

failure, where agents communicate sporadically, or when they must deal with disconnection periods to save battery. This situation appears in rural areas, characterized by limited connectivity and where the system may remain isolated without supervision for extended periods. These features can be obtained if, instead of using a centralized approach, a fully distributed one is used, as is the one followed by a consensus algorithm according to Olfati-Saber and Murray [5].

This paper presents a *consensus-based learning algorithm* coined Co-Learning or CoL, trying to take advantage of a completely decentralized approach for an FL-like learning algorithm. Along with presenting the *CoL* algorithm definition and description, an actual implementation using SPADE agents [6] is provided.

SPADE (Smart Python Agent Development Environment) [6] is a framework for developing intelligent agents in Python. The communication layer uses XMPP (Extensible Messaging and Presence Protocol)[1] as an instant messaging protocol.

This platform has been used in different areas, especially in IoT [7]. The CoL implementation in SPADE takes advantage of the *Presence* feature of the XMPP so that it can detect when a neighbor agent decides not to go on being connected or fails its connection, not having to wait for a deadline to acknowledge those failures. There is also some previous work in implementing a *pure* FL algorithm in *SPADE* agents, called *FLaMAS* [8].

In multi-agent systems, communication between agents is essential, and SPADE agents have an integrated message dispatcher that allows communication between them.

The SPADE agent model is based on behaviors. They are tasks that repeat upon a particular time pattern: one-shot, periodic, finite state machines (FSM) or even BDI (Belief Desire Intention) [9] behaviors, which allows reactive and deliberative capabilities in the agent.

The paper also presents a new Intelligent Virtual Environment (IVE) [10] generator, developed to test SPADE agents in a close-to-real-world scenario before deployment. Graphical simulations have always been a way of testing and validating applications (like in the work by Ikidid et al. [11]) where a simulation in ANYLOGIC is used to validate a model to control and fluidize vehicle traffic in a multi-intersection network). Checking qualitatively if a simulation seems to work correctly can save hours of work analyzing boring tables of numbers. The main problem with these simulations is that they usually cost a lot to build or even tune for a specific algorithm.

There is no novelty in proposing just another simulation framework, even if discussing a simulated environment, simply to deploy a Multi-Agent System. Traditionally, simulators that include agency concepts simulate the environment and the agents. That is the case, for instance, of *Netlogo* [12], where agents inhabit a matrix-like environment formed by patches. However, this simulator is limited to four different types of agents, the simulated environment is two-dimensional and does not allow the decoupling of its parts. The configuration of this monolithic system is produced in the same file.

On the other hand, it may be found what is called an IVE (Intelligent Virtual Environment) [10] that is, a virtual environment simulating a real world, inhabited by intelligent agents who may interact and whose behavior can be easily validated.

JaCalIVE (Jason Cartago implemented Intelligent Virtual Environment) [13] can be seen as an example of a framework for developing MAS inhabiting an IVE. This framework is based on MAM5 meta-model [14]. The idea behind it is to define a simulation through such a meta-model, which is compiled into some templates of Jason agents [15] and CaRTago artifacts [16] to be completed by the simulation developer. This framework has a very formal development process, but it is difficult to develop a new simulation or even make changes to an existing one.

It can also be found *MASON*(Multi-Agent Simulator Of Neighborhoods) [17], made purely in the Java programming language and released in 2003. This simulator is mainly oriented toward swarm intelligence and multi-agent systems. In addition, it allows you to choose a discrete or continuous space in the simulations and visualize the result in a two or three-dimensional space. However, achieving a 3D visualization in this simulator is not easy or fast and requires the additional installation of the Java3D libraries and knowledge of Java programming.

Differently, the proposal presented in this paper looks for an easy way of defining and modifying an IVE. This IVE will be developed in Unity[2], and agents will be SPADE agents [6]. This Simulation framework, named *FIVE* [18], allows us to define the environment and incorporate the algorithms to be validated into SPADE agents inhabiting such an environment.

The rest of the paper is structured as follows: in Section II the Co-Learning algorithm is presented. Next, in Section III the implementation of this algorithm in SPADE agents is shown. After that, in Section IV new FIVE framework is presented as a way of easy and fast creation and modification of Intelligent Virtual Environments to test Co-Learning SPADE agents, followed by Section V, where a case study with a virtual environment simulating an orange orchard is presented. The paper ends with some conclusions in Section VI.

II. Co-Learning (CoL) Algorithm

This section presents the model that supports the distributed training of the machine learning model, combined with the consensus process to average the parameters learned by the agents. An interaction topology delimits the ability of the agents to communicate and exchange information.

A. Consensus-Based Multi-Agent Systems

Olfati–Saber and Murray [5] define a consensus process in a Multi-Agent System (MAS) as a problem where the agents reach an agreement about the value of a variable of interest without any intermediate or leader that rules the process. It is an iterative procedure. The agent a_i calculates the new value $x_i(t+1)$ in each iteration, according to Equation (1).

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

where N_i denotes the neighbors of agent a_i and ε is the learning step: a factor bounded by the maximum degree of the network. The consensus converges to the average of the initial values $\langle x_i(0)\rangle$ whenever $\varepsilon \leqslant \frac{1}{\max d_i}$. This algorithm has been the base for different and multiple applications and other algorithms as, for instance, *Supportive Consensus* [19].

Fig. 1 depicts an example of the evolution of consensus over one of the weights over this simple synthetic network with four agents and initial values $x(0) = \{0.2, 0.4, 0.6, 0.8\}$. The convergence value is $\langle x(0)\rangle = 0.5$.

B. Consensus in Federated Learning

One of the approaches of FL consists of a set of clients that learns the weights of an artificial neural network (ANN) and shares them with a central server, which averages the weights to obtain a global model. Without losing generality, we can consider each weight as an independent variable and execute the consensus process in parallel over all the weights simultaneously.

[1] https://xmpp.org/

[2] http://unity.com

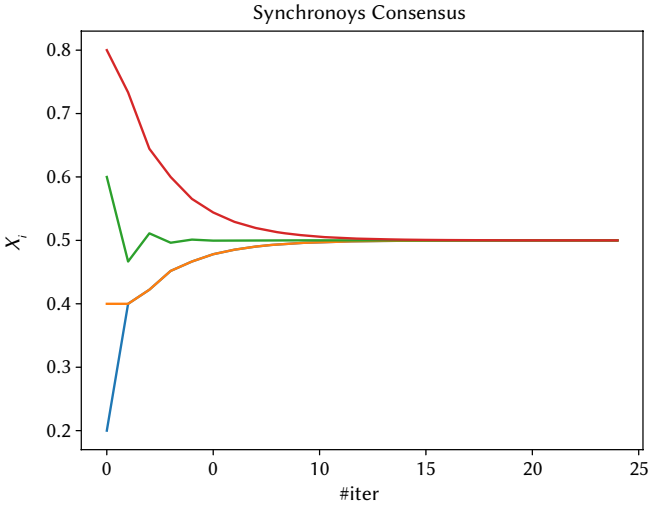


Fig. 1. Consensus evolution in a network with four agents. Initially, $x(0) = \{0.2, 0.4, 0.6, 0.8\}$, so $\langle x(0) \rangle = 0.5$.

Let us define a set of n identical agents A; each one implements identical ANN structures (same blocks, layers, and neurons). The goal is to learn a global model (W, tr) with a set of weights W for a training dataset tr. As the model is common, agents need only to share the set of weights W. The training dataset is divided into n fragments of the same length. The extrapolation of this approach to non-independent and identically distributed (non-IID) datasets is direct by using a weighted consensus variation [20].

The communication among the set of agents is constrained by a topology modeled by an undirected network $G = (A, E)$, where the nodes are the agents of set A. The set of edges E formed by pairs (a_i, a_j), denoting that agent a_i is connected with agent a_j. The neighborhood of agent a_i is denoted with $N_i = \{a_j \in A : (a_i, a_j) \in E\}$.

Each agent keeps an ANN (W_i, tr_i), being W_i a set of weights and biases for each layer of the ANN.

$$W_i = (W_{i,1}, \ldots, W_{i,k}) \tag{2}$$

where $W_{i,j} \in \mathbb{R}^{n,m}$ represents the weights (or the bias) learned by agent a_i for the layer j of its ANN. Without losing generality, we can assume that the parameters of the ANN can be reshaped into a conforming representation.

The process follows the adapted Equation (3).

$$W_i(t+1) = W_i(t) + \varepsilon \sum_{a_j \in N_i} [W_j(t) - W_i(t)] \tag{3}$$

C. Algorithm Description

The *Consensus-based Learning Algorithm*, named either *Co-Learning* or *CoL* Algorithm can be described as a set of identical agents learning a model through an ANN, where all the agents share the same ANN structure. This allows sharing the model being learned by each agent with its local neighbors and making a consensus of such model based on the Equation (1). This model is formed by the weights matrices result of the training of the learning process -Equation (2)-. This consensual model is then used for each agent in the next training. An agent a_i following the Co-Learning algorithm (Algorithm 1) first of all will make e epochs of training the algorithm. The result of this training is the set of k matrices at Equation (2), and for each one of them, the next c iterations of the consensus algorithm, following the Equation (1) are made, leading to k new matrices that will be used in the training process again.

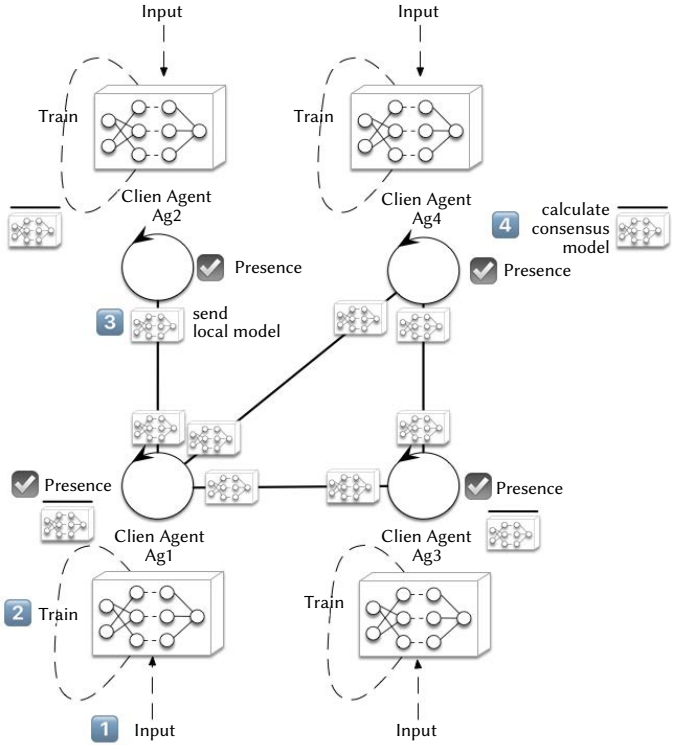


Fig. 2. Scheme of four SPADE agents doing a CoL Algorithm.

The process is executed in parallel as many times as parameters the ANN has. It can be considered a vectorized version of the evolution seen in Fig. 1.

Algorithm 1. Co-Learning (CoL) Algorithm for agent a_i

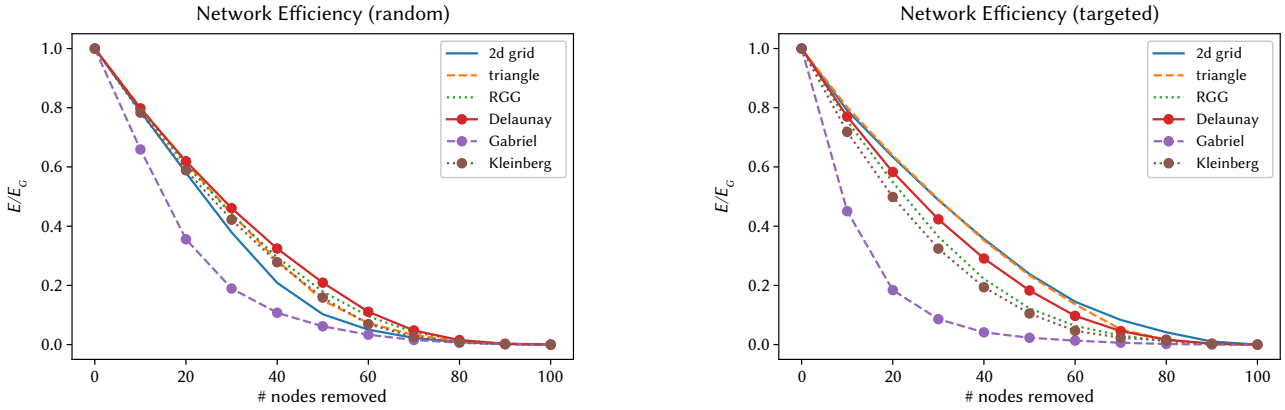
1: **while** !doomsday **do**
2: **for** $f \leftarrow 1, e$ **do**
3: $W \leftarrow Train(f)$
4: **end for**
5: **for** $j \leftarrow 1, k$ **do**
6: $X_i(0) \leftarrow W_j$
7: **for** $t \leftarrow 1, c$ **do**
8: $X_i(t+1) \leftarrow X_i(t) + \varepsilon \sum_{a_j \in N_i} [X_j(t) - X_i(t)]$
9: **end for**
10: **end for**
11: **end while**

D. Network Topology

The underlying network topology does not affect the final consensus value but does the convergence speed. The effect of different network structures has been studied by Carrascosa et al. [1]. Random geometric graphs (RGG) are the most balanced solution between the efficiency in achieving the consensus value and the robustness under deliberate or accidental failures.

In an RGG, agents are located randomly in a square-unit area and linked with neighbors within a determined radius. It's the equivalent of a random graph, considering the spatial location of the agents.

Fig. 3 shows the robustness to agent failures of different network topologies: square and triangular grids, Kleinberg's networks, RGG, Delaunay triangulation, and Gabriel's graph (a simplification of Delaunay one). Comparing random failures and deliberate attacks, RGG and Delaunay have an adequate balance between algorithm



(a) Efficiency under random failures

(b) Efficiency under deliberated attack

Fig. 3. Efficiency of different network topologies.

performance and resilience. Nevertheless, RGG scales better when the size of the network grows.

Therefore, the underlying structure selected to configure the acquaintance's graph is an RGG using a given radius from the initial location of the agents. When no spatial information about the agents is available, we distribute them randomly in a fictitious space.

III. Execution Using SPADE Agents

This section presents the CoL algorithm implementation using *SPADE* agents [21]. Fig. 2 shows the *Co-Learning algorithm* in a network formed by four SPADE agents. Our CoL system is composed of two types of agents, *initialization* and *learning* agents: There is one *initialization* agent in the platform and *n learning* agents.

As its name suggests, the *initialization* agent is the agent in charge of setting up the whole system. It starts with reading a CSV file, which contains all the information related to the construction of the network of agents, indicating to each agent who is in contact. So, each agent will subscribe to the *presence* functionality of its neighbors, provided by the XMPP protocol features. The presence is a feature provided by the XMPP protocol to SPADE agents, enabling an agent to ascertain the status of other agents. This functionality is particularly valuable for determining whether an agent is connected and available for information exchange. The *initialization* agent is a utility agent that is not involved in the consensus process (in fact, the system has been tested adding new SPADE agents to the process during the execution of the system, without using this *initialization* agent).

Learning agents carry out the CoL process, exchanging the ANN model information with the neighbors with mutual subscriptions. The behavior of these *learning* agents is defined as a finite-state machine (FSM) in *SPADE* (See Fig. 4).

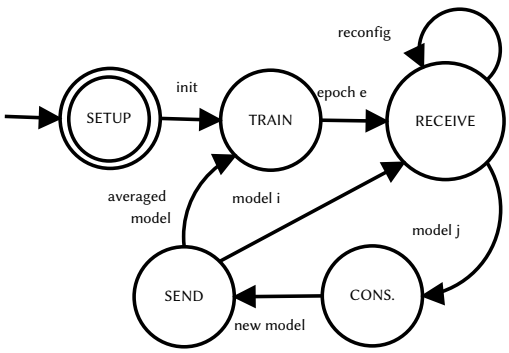


Fig. 4. FSM behavior for the SPADE learning agents doing the *CoL* Algorithm.

The first state is the *SETUP* state. In this state, the FSM that controls the agent is initialized. Then, it will pass to the *TRAINING* state, where it will train the model during e epochs. The next stage is the *RECEIVING* state, in charge of receiving two different kinds of messages: configuration messages and new training weights messages. The first one allows modifying the agent's acquaintances to change the network's structure if necessary. The second one is the messages the agents send to their neighbors during the consensus process to share their model. When the agent has received a message from all its active neighbors sharing their new training weights, it will pass to the *CONSENSUATING* state, where it will calculate a new aggregated model applying the consensus equation. Then, it will progress to the *SENDING* state, sending the latest model to its neighbors. While it is making c iterations of the consensus algorithm, it will go back to the *RECEIVING* state. When it has finished the c consensus iterations, it will go back to the *TRAINING* set, where it will use the new aggregated model to go on training during e epochs.

The *RECEIVE–CONS–SEND* loop finishes when there are no significant differences in the models. Then, the agent can begin a new training iteration if needed or conclude the complete process and use the ANN model.

IV. FIVE Framework

SPADE agents can run over the physical system or on a simulated one without relevant differences. Having available 3D virtual environments as close to reality eases the MAS' development and debugging, testing the agents' behavior in real-world conditions. This section describes the FIVE framework (Flexible Intelligent Virtual Environment developing framework) that will support the agents executing the CoL process.

A. FIVE Architecture

The FIVE framework is composed of three elements:

- The XMPP server.
- The FIVE simulator server, made with Unity[3].
- A set of SPADE agents that will populate the simulated environment.

Each component can transparently run on separate machines (including, of course, each SPADE agent being executed in a different host).

Fig. 5 shows an example of a FIVE simulation deployed into four local networks. The colored rectangles represent different local networks, and the arrows are network sockets. Each intelligent agent

[3] https://unity.com

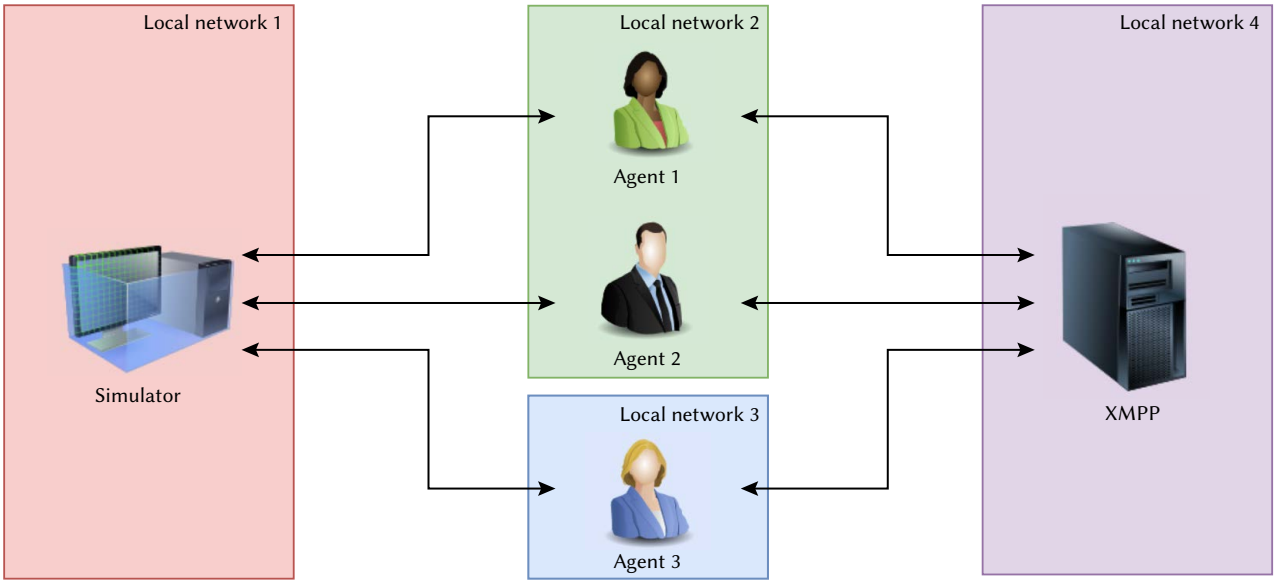


Fig. 5. Example of a FIVE simulation deployed in four different local networks.

represented in the figure runs on a different machine. *Agent 1* and *Agent 2* are on the same local network (*Network 2*). The three agents are connected to both the simulator and the XMPP server.

The FIVE simulator is a new tool made with Unity designed to define IVEs inhabited by SPADE agents. FIVE allows the creation of three-dimensional environments using a built-in text-based map editor. In addition, it will enable the rapid creation of custom agent avatars equipped with sensors, such as a camera.

FIVE agents (based on SPADE) control the virtual avatar in the IVE managed by the simulator. The framework grants network failure toleration: if an agent is disconnected from the FIVE simulator, it can be reconnected easily and resume its activity.

B. Defining a Simulation

FIVE simulations are composed of the environment created by the simulator and the intelligent agents that inhabit it. Defining a simulation is a process that just involves four text files (see Fig. 6). Three define the environment with elements such as terrain, trees, or light conditions, and the last file is used to create the agents.

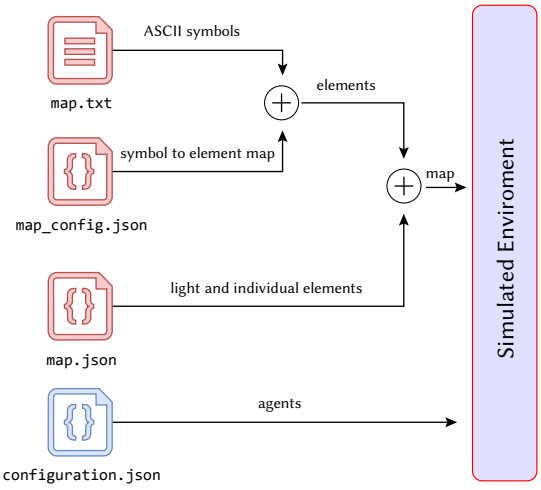


Fig. 6. FIVE simulator environment and agent generation from input files. The first three red files generate the intelligent virtual environment (composed of light objects, agent spawn points, and other elements), and the last blue file is used to generate the agents.

The file named `map.txt` is a text-based map where each ASCII character represents an object in the environment (Fig. 11 contains examples of all these files for the case study). This design decision was made to create the simulations easily and even modify the file through a text-based console.

The second file, `map_config.json`, assigns the map letters in the `map.txt` file to the objects in the simulator. The configurable properties are:

- `origin`: It represents the starting point where the elements will be placed into the simulation as a three-dimensional coordinate.
- `distance`: It represents the separation distance between elements in the different axis.
- `symbolToPrefabMap`: It is a list formed by three elements: the character that represents the element in map.txt file, the reference name of the object that replaces the letter, and an optional path that contain images to represent the object in the simulator

The third file used to generate the environment is `map.json`. The file sets the environmental conditions, such as light objects, and configures individual special elements. For example, if we need a river and a bridge that connects the two sides, the file includes configurable properties for these objects. The file contains two lists: one for objects with light properties and another for objects that do not need them. The main configurable properties are:

- `active`: Flag to create the object or ignore it.
- `objectName`: Internal name of the object.
- `position`: 3d coordinate where the element will be placed.
- `rotation`: Rotation (in degrees) in the three axes.
- `color`: Object with color data, in RGB and an alpha channel for opacity.
- `intensity`: Intensity of light ray.

Besides the files for generating the environment, the `configuration.json` file generates the inhabitant agents. This file includes the definition of all the information needed by the agents, including the FIVE simulator IP address, the avatar of the agents, or the spawn position. The configurable properties for the agents are:

- `name`: Name of the agent.
- `at`: XMPP server direction.

(a) Simulation of one tractor agent in an orange orchard field

(b) Simulation of four tractor agents and one robot agent in an orange orchard field

Fig. 7. Example of a simulation of an orange orchard field and agents. (a) there is only one agent. (b) there are five agents, and the space between trees is three times smaller than in figure (a).

- `imageBufferSize`: Maximum number of images per agent.
- `imageFolderName`: Name of the folder where images are saved. (related to the images perceived by the camera of the agent).
- `enableAgentCollision`: If this value is set to true, this agent will collide with other agents. Otherwise, it won't.
- `prefabName`: Avatar reference name for the agent.
- `position`: Spawn point position that can be referenced by name or by three-dimension coordinate.

FIVE simulator includes a library of existing elements by default, which can be incremented with new imported models. It contains several agent avatars that can be assigned to any inhabitant agent. Additional agent avatars can be added to the simulator through the Unity editor. The same can be said about the remaining objects that can be used to define the IVE.

It is important to underline that agent avatars include a configurable camera component so that the agent can take pictures of the IVE. Users can follow the track of any agent in the IVE in a first-person view. The resolution of these images can be easily configured. The camera component is not exclusive to included avatars; newly designed ones can also incorporate it.

To illustrate the effect of the change in configuration files, Fig. 7 depicts two different scenarios. In Fig. 7a, the distance between trees is nine, and there is only one agent in `configuration.json`. For Fig. 7b, we have reduced the distance to three, with five agents in the correspondent file.

C. Agents Programming

After defining the IVE, the next step is to program the agents' behavior. The FIVE framework includes an inhabitant agent's template, formed by a generic SPADE agent with an FSM behavior that implements the agent's execution cycle for communications with the

IVE. The code is addressed to control the avatars in the environment. The rest of the cognition related to the domain is included in the normal SPADE behaviors. The execution cycle (see Fig. 8) is composed of the following four states:

1. INITIAL STATE: The agent initializes variables to be referenced in other states. It also starts an instance of the *ImageManager* class on a background thread. The *ImageManager* class handles the incoming stream of captures taken by the agent's avatar in the FIVE simulator. It also adds the image data to a shared thread-safe queue for further processing.
2. PERCEPTION STATE: This state captures the image queue, dequeues them, and passes them to the agent behavior so that the images can be used for further process. The pictures are also automatically stored in the file system if desired.
3. COGNITION STATE: This state is where the process of cognition occurs. The agent decides to perform an action based on the information that it has at the moment.
4. ACTION STATE: In this state, the agent sends commands to the agent's avatar in the simulator. An example could be a camera rotation command or a move command.

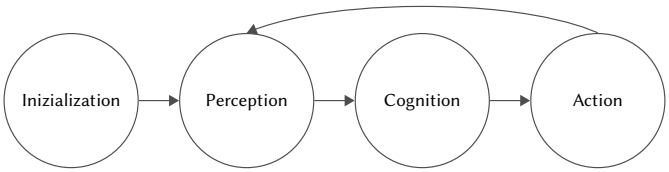


Fig. 8. FIVE agent FSM to control the avatar in the IVE.

Agent programming is done in a file named `entity_shell.py`. This file is an abstraction of the agent behavior explained above. It contains four methods that can be overloaded: `init`, `perception`, `cognition`, and `action`. Each method controls the execution of the agent in the corresponding FSM state.

The agent has access to a `Commander` class which defines an abstraction layer with the FIVE network protocol and contains methods to ease communication with the FIVE simulator. The current commands covered by `Commander` are:

- `create_agent`: It sends an instantiate request to the simulator, and the simulator returns the starting position coordinates to the agent. This command is always sent during the initial state to create the agent's avatar.
- `move_agent`: It sends a command to the simulator to move the agent's avatar to the desired position defined as (`x`, `y`, `z`). The simulator returns the agent the target position if the agent's avatar can reach this position. In the other case, the simulator returns the location where the agent got stuck.
- `fov_camera`: It sends a command to change the field of view value of the camera.
- `move_camera`: It sends a command to move the camera position.
- `rotate_camera`: It sends a command to rotate the camera.
- `take_image`: controls the image capture from the IVE.
- `change_color`: It sends a command to change the color of the agent.

Fig. 9 shows a possible execution interaction between one inhabitant agent and the FIVE simulator. The agent sends a first message to initiate the avatar in the simulation, adjusts the camera, and tries to move across the environment.

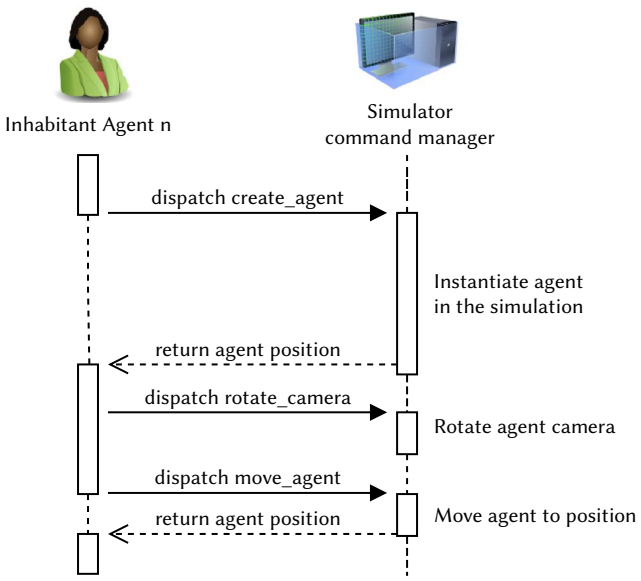


Fig. 9. FIVE agent communication through XMPP messages.

D. Executing a Simulation

With all the previous elements set, the FIVE system is ready for execution. It starts with the FIVE simulator generating the map elements, such as lighting, trees, or walls, and locating agents at their spawn points. First, the FIVE simulator parses the file named `map.json` and places the elements described by the JSON file in the IVE. Then, the FIVE simulator processes the ASCII characters in `map.txt`, situating the corresponding pieces in the IVE according to the letters' definitions. The simulator parses the file map_config.json to get the letters' associations and also sets the origin position for placing the elements and the amount of space between items.

Once the environment is ready, the FIVE simulator listens for incoming requests, handling the recently created sockets in new threads. The simulator provides the starting position coordinates as an answer to any agent sending a `create_agent` command, indicating its entity type information and spawning location data. The agent then initiates a new thread to handle the image socket's data reception to keep synchronized with the avatar and the agent.

Finally, each agent starts the FSM behavior that loops over the `perception`, `cognition` , and `action` states. The simulator process and executes all the commands, reflecting the agent actions in the IVE.

V. Case Study: A Simulation of an Orange Orchard Smart Area

The case study consists of the simulation of an IVE modeling an orange grove smart area. This simulation aims to test the CoL algorithm to train an ANN capable of detecting fruit diseases and what kind of disease it is. Several robots patrol the fruit orchard. Each robot trains its ANN with pictures of the fruits it views. Once the individual models are trained, they are shared and aggregated by consensus with CoL. The result is a model trained with the complete image dataset, even with pictures a particular robot has never seen.

This case uses several of the three-dimensional models available within the FIVE simulator: a tractor robot, a tree, and a white box representing the fruit. The white boxes will have orange textures that will be loaded from `map_config.json` `dataFolder` path, so the trees will show actual oranges hanging on their branches. Fig. 11 depicts some details of the configuration files with the map and object characteristics.

Fig. 10. Agents *agente1* (yellow tractor) and *agente2* (red tractor) patrolling and taking pictures of the oranges in the grove. Notice the "Tree Fruit Variant" trees with random textures of orange fruits applied at runtime, loading the images from the folder path specified in `map_config.json` file.

```
A A
   O G O G O
   O G O G O
   O G O G O
   O G O G O
   O G O G O
   O G O G O
   O G O G O
   O G O G O
   O G O G O
   O G O G O
```

(a) Portion of `map.txt` content.

```
{
   " symbol ": "G",
   " prefabName ": " Tree ␣ Fruit ␣ Variant ",
   " dataFolder ": "C:/ oranges / green "
},
{
   " symbol ": "A",
   " prefabName ": " Spawner "
}
```

(b) Portion of `map_config.json` content.

```
" active ": true ,
" objectName ": " Tree ␣ 1",
" objectPrefabName ": " Tree ",
" position ": {
  "x": -2.6,
  "y": 0.0,
  "z": 0.0
},
" rotation ": {
  "x": 0.0,
  "y": 5.0,
  "z": 0.0
```

(c) Portion of `map.json` content.

```
{
   " name ": " agente1 ",
   "at": " localhost ",
   " password ": " xmppserver ",
   " imageBufferSize ": 3,
   " imageFolderName ": " captures ",
   " enableAgentCollision ": true ,
   " prefabName ": " Tractor ",
   " position ": " Spawner ␣ 1"
}
```

(d) Portion of `configuration.json` content.

Fig. 11. Portions of the content of the four different files involved in defining the FIVE IVE of the case of study.

1. The letters in the map represents: *A* letter is replaced by an agent spawner point, *O* and *G* characters are orange trees. The difference is that *G* trees only have green oranges.
2. Besides identifying *G* and *O* with trees, the file contains in the `dataFolder` fields the paths with the corresponding orange pictures depending on its color. The orange ones might include diseases.
3. The environmental conditions include an isolated with a custom position and rotation in this case.
4. The last file includes the agents' declaration. Its `position` property refers to the name of the (invisible object) spawner where the agent will be created.

When the simulation starts, `map.json` file is parsed and its elements (the light and an isolated tree without oranges) are placed into the environment. Then the other components (agent spawner points and orange trees) from `map.txt` are added to the scene using the `map_config.json` information. Finally, the agents described in `configuration.json` file are spawned in the simulation and walk through the grove field, taking images of the oranges (see Fig. 10).

Although it can be specified in other ways, and even personalized in different ways for each agents, the test made have considered a random network.

A. Disease Identification

To validate the simulator, agents integrate a plant disease classification ANN. The architecture used for the experiments was a Mobilenet V2 [22] with the following hyperparameters definition: the agents make one epoch in their training step before changing to the receiving state in the FSM machine. The models of all agents are identical, having undergone training using data augmentation and fine-tuning, employing the following set of hyperparameters: Global Epochs:1; Local Epochs: 10; Local Batch Size: 10; Learning Rate: 0.001; SGD momentum: 0.5; Number of Each Kind of Kernel: 9; Number of Filters for Conv Nets: 32; Max Pooling: Yes; Network: CNN; Transfer learning (Tl): Yes or No.

This network was trained using the dataset presented in port [23], which has four classes Blackspot, Canker, Fresh, and Grenning. The dataset is divided into 80% for training, 10% for validation and 10% for testing. The training set contains 207 images for Blackspot, 202 images for Canker, 389 images for Fresh, and 370 images for Grenning. The testing set contains 139 images for Blackspot, 149 images for Canker, 165 images for Fresh, and 177 images for Grenning. Fig. 12 shows some pictures extracted from the dataset used to perform the training. The dataset images have been distributed along the different orange trees in the simulation, and each agent is able to access only a subset of trees as they are distributed along the different parts of the orange orchard. So, they are using different parts of the dataset.

Fig. 12. Four sample images, one image of each citrus dataset class.

Fig. 13 shows the accumulated accuracy and loss obtained in the training process and the confusion matrix is presented and elaborated in the "Execution using SPADE agents" section of the original article, where a convergence analysis of the CoL algorithm have been conducted [1]. After training the network using the CoL process, the obtained model was integrated into the `Cognition` method available for the inhabitant agents and used for testing the model against the testing part of the dataset commented above.

As commented above, these agents include, by default, a camera for capturing images. The camera was adjusted using the `Commander` API, modifying its position and field-of-view via commands to focus on the fruit images as the tractor robot moved along the grove. These fruits were images of fruits loaded according to the dataset path indicated in the `map_config.json` file. Executing the agents would allow validating the values we obtained when training the network.

B. Modifying the Field Configuration

In this section, we are going to make modifications to the case of study in order to illustrate how simple it is to change a simulation in the FIVE framework. The modification consists of dividing the trees into five classes. As we have more agents capable of identifying diseases of the orange grove, we will obtain faster identification. Each agent will be spawned in a different column. Therefore, it will only be necessary for everyone to go through their column once to obtain captures of all the trees in the orchard. We are also going to modify the environmental conditions so that the captures are taken at night, checking the identification precision under poor light conditions.

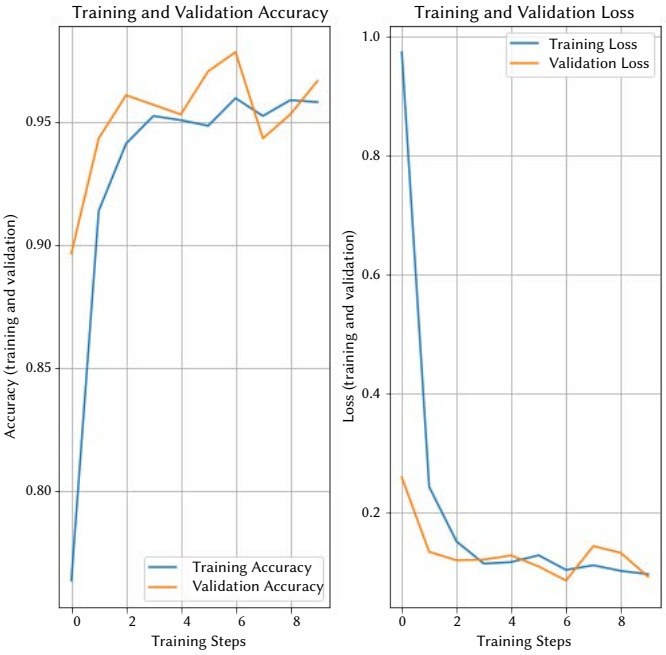


Fig. 13. Accumulated Train Accuracy and Accumulated Train Loss.

First, we have to create five folders: one folder for each class. Each folder will contain four images rendered as a texture and randomly applied to the oranges in the tree to which the folder class belongs. To achieve this, we must modify the `map.txt` and `map_config.json` files. In `map.txt`, we add three more *A* letters to create the new agents, and we have to define a character for each tree class. For example, we can use *B* for black spot, *C* for Canker, *G* for Greening, *M* for Melanose, and *H* for Healthy. Then, in map_config.json file, we match the characters with the elements they represent, as letter *G* is defined in Fig. 11b. Finally, we must modify the `dataFolder` property with the folder where the images are to load the textures.

Next, we must update the `map.json` file to modify the environmental conditions. Our desired light condition is moonlight, so we can change the intensity and color of the light used in map.json without writing a single line of code.

Finally, we have to modify configuration.json file and add three more agents as *agente1* is defined in Fig. 11d. We can change the `name` property of the new agents to *agente3*, *agente4*, and *agente5*. We can also set the initial position of them in *Spawner 3*, *Spawner 4*, and *Spawner 5* generation points. Lastly, it has to be indicated the neighbors of the new agents generated.

Once we have defined all our modifications in the four files involved, we can start the simulation process, and the result is shown in Fig. 14. As a result, we have a completely new environment to test whenever the ANNs trained in good light conditions are valid or if they need some retraining process to adjust the parameters to the new scenario.

C. FIVE Loading Time Test

The last experiments measure the load time FIVE simulator needs to load a complete scene populated by elements that use the FIVE system to apply textures from images at runtime.

The dimension of the images for the textures is 224 × 224 pixels, randomly chosen from five classes located in folders with four images each, adding a total of twenty different pictures. The execution platform is a laptop without an external graphics card and with the following components: an integrated graphics model *Intel Iris Plus Graphics*, an *Intel Core i7 processor 1035G7*, 16 GB of RAM, a motherboard *ASUSTeK X421JAY* and a storage device *NVMe Intel SSDPEKNW01*.

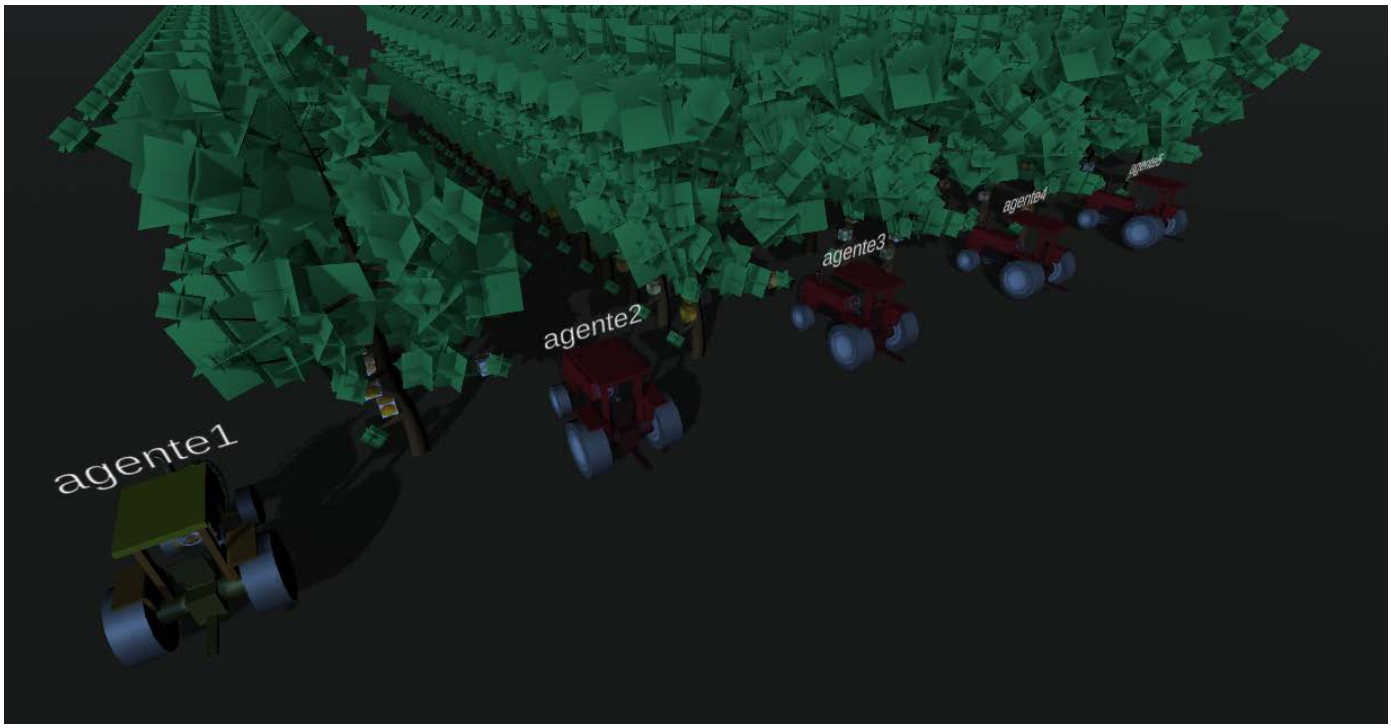


Fig. 14. Five agents in the modified case of study with nocturnal environmental conditions.

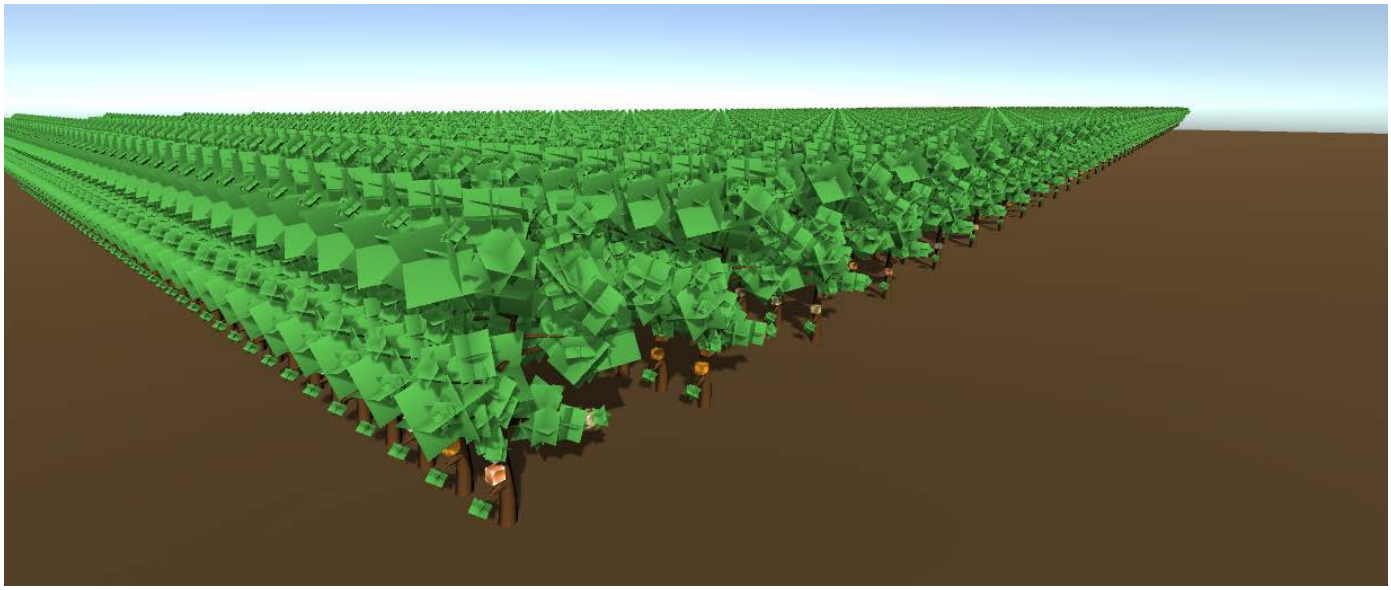


Fig. 15. Modified case of study with daylight environment conditions and two thousand five hundred orange trees.

Fig. 16 shows a graph that illustrates the time (in seconds) it has taken to load the entire scene, populating it with a light object, the terrain, and a variable number of fruit trees. The loading time of the whole scene has been measured, not just the texture loading process.

In conclusion, we can see that FIVE is ready to load complex environments quickly and effectively. The reason is that FIVE uses optimization techniques that allow us to simulate scenes with a large number of different elements.

VI. Conclusions

We have presented a Consensus-based Learning algorithm (CoL) that takes advantage of distributed learning based on the idea behind federated learning of sharing a model between a set of agents. This advantage is based on complementing individual models the agents train with their aggregation. By doing this, all agents may benefit from the training completed by the rest of the agents. The agents share the parameters of the models but not the training data. Therefore, privacy is maintained during the training. As we use a consensus-like algorithm for the model's aggregation, we have some other advantages as the adaptation to variations in the agent set, allowing agents to abandon and enter during the execution. The paper shows the implementation of CoL algorithm in *SPADE* agents.

RGG topology improves the performance of the convergence of consensus since the average path lengths are shorter than the rest of the networks and is a pretty robust topology under random or deliberate failures. Therefore, we propose its use as the underlying structure for the MAS.

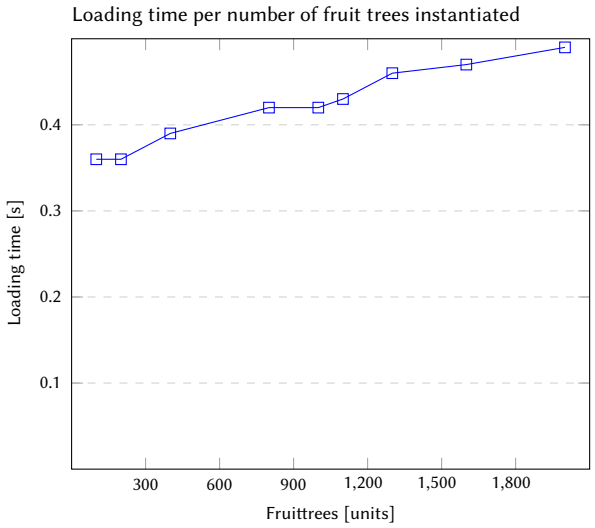


Fig. 16. Graph showing the time it took to load the scene composed of fruit trees, with four fruits each, and loading their textures at runtime from images.

Moreover, we have presented a new framework called FIVE that allows the easy creation and modification of IVEs inhabited by SPADE agents. This framework has been used to test CoL in SPADE agents through an orange orchad simulation.

As part of our future work, we are dedicated to enhancing the communication between agents in the CoL process. This includes optimizing message transmission, both in terms of quantity and size. Additionally, we are actively exploring the generation of simulated maps, where satellite images are leveraged to create them automatically. Lastly, we are delving into the possibility of introducing semantic coalitions among agents. This entails agents that share similar meanings in the data they handle, engaging in more frequent information exchange with each other compared to other agents in the network.

Acknowledgment

This work has been developed thanks to the funding of projects:

- Grant PID2021-123673OB-C31 funded by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe"
- PROMETEO CIPROM/2021/077
- TED2021-131295B-C32
- Ayudas del Vicerrectorado de Investigacion de la UPV (PAID-PD-22)

References

[1] C. Carrascosa, J. Rincón, M. Rebollo, "Co-learning: Consensus-based learning for multi-agent systems," in *Advances in Practical Applications of Agents, Multi-Agent Systems, and Complex Systems Simulation. The PAAMS Collection*, 2022, pp. 63–75.

[2] H. Sánchez San Blas, A. Carmona Balea, A. Sales, L. Augusto Silva, G. Villarrubia González, "A platform for swimming pool detection and legal verification using a multi-agent system and remote image sensing," *International Journal of Interactive Multimedia and Artificial Intelligence*, 2023, pp. 1-13, doi: 10.9781/ijimai.2023.01.002.

[3] H. Brendan McMahan, E. Moore, D. Ramage, S. Hampson, B. Agüera y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*, 2017, pp. 1273–1282, PMLR.

[4] P. Kairouz, H. McMahan, B. Avent, A. Bellet, M. Bennis, A. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, et al., "Advances and open problems in federated learning," *Foundations and Trends in ML*, vol. 14, no. 1–2, pp. 1–210, 2021.

[5] R. Olfati-Saber, R. M. Murray, "Consensus problems in networks of agents with switching topology and time-delays," *IEEE TAC*, vol. 49, no. 9, pp. 1520–1533, 2004.

[6] J. Palanca, A. Terrasa, V. Julian, C. Carrascosa, "SPADE 3: Supporting the new generation of multi-agent systems," *IEEE Access*, vol. 8, pp. 182537–182549, 2020, doi: 10.1109/ACCESS.2020.3027357.

[7] J. Palanca, J. Rincon, V. Julian, C. Carrascosa, A. Terrasa, "Developing iot artifacts in a mas platform," *Electronics*, vol. 11, no. 4, p. 655, 2022.

[8] J. Rincon, V. Julian, C. Carrascosa, "Flamas: Federated learning based on a spade mas," *Applied Sciences*, vol. 12, no. 7, pp. 1–14, 2022, doi: 10.3390/app12073701.

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

[10] M. Luck, R. Aylett, "Applying artificial intelligence to virtual reality: Intelligent virtual environments," *Applied artificial intelligence*, vol. 14, no. 1, pp. 3–32, 2000.

[11] A. Ikidid, E. F. Abdelaziz, M. Sadgal, "Multi-agent and fuzzy inference-based framework for traffic light optimization," *International Journal of Interactive Multimedia and Artificial Intelligence*, vol. 8, no. 2, pp. 88-97, 2023, doi: 10.9781/ijimai.2021.12.002.

[12] U. Wilensky, "Netlogo (and netlogo user manual)," *Center for connected learning and computer-based modeling, Northwestern University*. http://ccl. northwestern. edu/netlogo, 1999.

[13] J. Rincon, E. Garcia, V. Julian, C. Carrascosa, "The jacalive framework for mas in ive: A case study in evolving modular robotics," *Neurocomputing*, vol. 275, pp. 608–617, 2018.

[14] A. Barella, A. Ricci, O. Boissier, C. Carrascosa, "Mam5: multi-agent model for intelligent virtual environments," in *10th european workshop on multi-agent systems (EUMAS 2012)*, 2012, pp. 16–30.

[15] R. H. Bordini, J. F. Hübner, M. Wooldridge, *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons, 2007.

[16] A. Ricci, M. Viroli, A. Omicini, "Cartago: A framework for prototyping artifact-based environments in mas," in *International Workshop on Environments for Multi-Agent Systems*, 2006, pp. 67–86, Springer.

[17] S. Luke, G. C. Balan, L. Panait, C. Cioffi-Revilla, S. Paus, "Mason: A java multi-agent simulation library," in *Proceedings of Agent 2003 Conference on Challenges in Social Simulation*, vol. 9, 2003.

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

[19] A. Palomares, M. Rebollo, C. Carrascosa, "Supportive consensus," *PLOS ONE*, vol. 15, no. 12, pp. 1–30, 2020.

[20] F. Pedroche, M. Rebollo, C. Carrascosa, A. Palomares, "Convergence of weighted-average consensus for undirected graphs," *International Journal of Complex Systems in Science*, vol. 4, no. 1, pp. 13–16, 2014.

[21] J. Palanca, A. Terrasa, V. Julian, C. Carrascosa, "Spade 3: Supporting the new generation of multi-agent systems," *IEEE Access*, vol. 8, pp. 182537–182549, 2020, doi: 10.1109/ACCESS.2020.3027357.

[22] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, L. Chen, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," *CoRR*, vol. abs/1801.04381, 2018.

[23] M. C. Silva, J. C. F. da Silva, R. A. R. Oliveira, "Idissc: Edge-computing-based intelligent diagnosis support system for citrus inspection.," in *ICEIS (1)*, 2021, pp. 685–692.

Carlos Carrascosa

Dr. Carlos Carrascosa was born in Valencia (Spain) and received the M.S. degree in Computer Science from the Universidad Politécnica de Valencia (UPV), in 1995. He obtained his Ph.D. in the Departamento de Sistemas Informáticos y Computación at UPV and is currently a Lecturer involved in teaching several AI-related subjects at the UPV. He is member of the VRAIn (Valencian Research Institute for Artificial Intelligence) where he develops his research that include MAS, Federated Learning, consensus in MAS, IVEs, social emotions, serious games, information retrieval, and real-time systems.

Francisco Enguix

Francisco Enguix Andrés was born in Valencia (Spain). He is pursuing a Master's degree in Artificial Intelligence, Pattern Recognition and Digital Imaging at Polytechnic University of Valencia (UPV) while he works at the Valencian Institute for Research in Artificial Intelligence (VRAIN). He holds a degree in Computer Science from the Polytechnic University of Valencia (UPV) since 2022.

Miguel Rebollo

Dr. Miguel Rebollo received his PhD. in Artificial Intelligence from the Universitat Politècnica de València (Spain) in (2004), and Dr. in Complex Systems from the Universidad Politècnica de Madrid (Spain) in 2019. He is a member of the Valencian Research group for Artificial Intelligence (vRAIN). He works as Associate Professor at the Universitat Politècnica de València. His research interests involve complex intelligent adaptive systems, multi-agent systems, chaos and non-linear systems, and social network analysis.

Jaime Andrés Rincón Arango

Jaime Andrés Rincón Arango is a postdoctoral researcher at the Valencian Institute for Research in Artificial Intelligence (VRAIN) of the Polytechnic University of Valencia. He holds a degree in Biomedical Engineering from the Universidad Manuela Beltrán (Colombia), a Master's degree in Artificial Intelligence from the Universidad Politécnica de Valencia and a PhD in Computer Science from the Universidad Politécnica de Valencia. His main research activities focus on IoT, IoMT, Cognitive Assistants, assistive robotics for the elderly and Edge AI. He is author or co-author of more than 50 articles in specialized journals and national and international conferences.