D5.1: Initial report on distributed AI and AI-based orchestration

Revision: v.1.1

Work package	WP5
Task	T5.1, T5.2, T5.3
Due date	29/Feb/2024
Submission date	29/Feb/2024
Deliverable lead	Dušan Jakovetić (UNS)
Version	1.1
Authors	Dušan Jakovetić (UNS), Lidija Fodor (UNS), Angela Agošton (UNS), Miroslav Popović (UNS), Miloš Simić (UNS), Claudia Soares (NOVA), Francisco M. Caldas (NOVA), Stevo Racković (NOVA), Frederico Metelo (NOVA), Sotiris Spantideas (NKUA), Ilias Paralikas (NKUA), Nobuko Yoshida (UOXF), Ping Hou (UOXF), Roland Kuhn (ACT), Giovanni Granato (GMV), Dimitra Tsigkari (TID), Alceste Scalas (DTU), Rafael Oliveira Rodrigues (EDP)
Internal reviewers	Dimitra Tsigkari (TID) Ping Hou (UOXF)

Grant Agreement No.: 101093006
Call: HORIZON-CL4-2022-DATA-01
Topic: HORIZON-CL4-2022-DATA-01-03
Type of action: HORIZON- RIA

Abstract	This document describes the initial advances in the lightweight and distributed AI primitives in T5.1 and T5.3, towards addressing the AI application needs as identified in D2.1, including results' refinements based on D2.2. The report also includes preliminary analyses and approaches on the AI-based orchestration in T5.2. In addition, initial machine learning modeling for the TaRDIS use cases has been provided.
Keywords	decentralized machine learning and inference; AI/ML programming primitives, AI-driven planning, deployment and orchestration; lightweight and energy efficient ML techniques

Document Revision History

Version	Date	Description of change	List of contributor(s)
V0.0	09/11/2023	Table of contents draft released.	UNS, NKUA, NOVA.
V1.0	16/02/2024	Document ready for internal review	all authors
V1.1	20/02/2024	Document reviewed internally	TID, UOXF

Disclaimer

Funded by the European Union

Funded by the European Union (TARDIS, 101093006). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

Copyright Notice

Project funded by the European Commission in the Horizon Europe Programme		
Nature of the deliverable:	**R**	
Dissemination Level		
PU	*Public, fully open, e.g. web (Deliverables flagged as public will be automatically published in CORDIS project's page)*	✔

SEN	*Sensitive, limited under the conditions of the Grant Agreement*	
Classified R-UE/ EU-R	*EU RESTRICTED under the Commission Decision No2015/ 444*	
Classified C-UE/ EU-C	*EU CONFIDENTIAL under the Commission Decision No2015/ 444*	
Classified S-UE/ EU-S	*EU SECRET under the Commission Decision No2015/ 444*	

* R: Document, report (excluding the periodic and final reports)

DEM: Demonstrator, pilot, prototype, plan designs

DEC: Websites, patents filing, press & media actions, videos, etc.

DATA: Data sets, microdata, etc.

DMP: Data management plan

ETHICS: Deliverables related to ethics issues.

SECURITY: Deliverables related to security issues

OTHER: Software, technical diagram, algorithms, models, etc.

Executive summary

The TaRDIS project aims to build a distributed programming toolbox that simplifies the development of decentralized, heterogeneous swarm applications deployed in diverse settings. WP5 involves three main objectives: the development of a framework that supports decentralized learning and inference by using artificial intelligence/machine learning (AI/ML) primitives, the use of reinforcement learning and other relevant approaches to exploit and specialize the preceding for planning, deployment and orchestration of the TaRDIS framework and the development of novel lightweight ML techniques that support decentralized and swarm learning in resource-constrained devices. Therefore, WP5 consists of 3 tasks: Task 5.1 Framework supporting AI/ML programming primitives, Task 5.2 AI-driven planning, deployment and orchestration framework and Task 5.3 Library of lightweight and energy efficient ML techniques.

The objective of this deliverable is to provide an initial report on the development of distributed AI/ML and AI-based orchestration, through the tasks under WP5, for the period of the first 14 months of the project. The deliverable provides a description of the positioning of the ML/AI tools in the overall TaRDIS framework through connections with other technical activities carried out in the project, the description of the TaRDIS use cases with the related proposed initial ML modeling approaches and a technical overview of the advances on the tasks under WP5. These advances include: the introduction and demonstration of the Python Testbed for Federated Learning Algorithms (PTB-FLA) framework, with possible TaRDIS usages, the description of the algorithms developed in Flower framework under FL training models APIs, a description of a simulation environment for reinforcement learning (RL) and RL agents using the simulator with an explanation of the developed Double Deep Q-Network (DDQN) algorithm, the description and positioning in TaRDIS of identified relevant techniques for making ML models more lightweight and energy efficient, and a description of further steps of development under the described tasks, where the future simulations are planned to be performed on real data. This report also contains the descriptions of the initial ML modeling approaches for all TaRDIS use cases and preliminary results.

Funded by the European Union

Table of contents

LIST OF FIGURES

Funded by the European Union

LIST OF TABLES

Abbreviations

ADMM	Alternating Direction Method of Multipliers
AI	Artificial Intelligence
API	Application Programmer Interfaces
BESS	Battery Energy Storage System
CLR	Centralized Logistic Regression
CNN	Convolutional Neural Network
DDPG	Deep Deterministic Policy Gradient
DDQN	Double Deep Q-Network
DL	Deep Learning
DNN	Deep Neural Network
DQN	Deep Q-Network
DRFL	Deep Reinforcement Federated Learning
DRL	Deep Reinforcement Learning
EKF	Extended Kalman Filter
ESS	Energy Storage System
EV	Electric Vehicle
FL	Federated Learning
FLA	Federated Learning Algorithm
FLaaS	Federated Learning as a Service
GAN	Generative Adversarial Network
GDPR	General Data Protection Regulation
GNSS	Global Navigation Satellite System
gRPC	google Remote Procedure Calls
HEMS	Home Energy Management System
HVAC	Heating, Ventilation and Air-Conditioning
IDE	Integrated Development Environment
ISL	Inter Satellite Link

IoT	Internet of Things
IPR	Intellectual Property Rights
JVM	Java Virtual Machine
KD	Knowledge distillation
LEO	Low Earth Orbit
LSTM	Long Short Term Memory
LTE	Learn To Exit
MDP	Markov Decision Process
ML	Machine Learning
MVP	Minimum Viable Product
mWTGs	micro Wind Turbine Generators
ODTS	Orbit Determination and Time Synchronization
P2P	Peer-To-Peer
PFedMe	Personalized Federated learning with Moreau envelopes
PTB-FLA	Python Testbed for Federated Learning Algorithms
PV	Photovoltaics
RF	Radio Frequency
RL	Reinforcement Learning
SNA	Social Network Ads
SPMD	Single Program Multiple Data

1 Introduction

1.1 Overview

Work package 5 is dedicated to the development of decentralized machine learning solutions. Its main objectives involve three directions, assigned to three tasks: creating a decentralized learning and inference framework supporting AI/ML primitives (Task 5.1), developing an AI-driven planning, deployment and orchestration framework (Task 5.2) and providing a library of lightweight and energy efficient ML techniques (Task 5.3). This document is the first report on ongoing work regarding WP5 and aims at demonstrating the progress of each task within WP5 for the period of the first 14 months of the project.

The decentralized ML solutions represent an important building element of the TaRDIS framework. Therefore, WP5 is closely related to different tasks from other work packages. Regarding WP1, the relation is presented in T1.1 through the continuous technical management of the project, in T1.2 mainly through continuous risk management, in T1.3 via innovation management and Intellectual Property Rights (IPR) and in T1.4 via General Data Protection Regulation (GDPR) and ethical AI guidance. The connection with WP2 can be identified by use cases AI/ML tasks analysis in T2.1, identification of AI/ML requirements in T2.2 and WP5 inputs to TaRDIS architecture in T2.3. The relation to WP3 is evident through programming models and abstractions in T3.1, Application Programmer Interfaces (APIs) for WP5 tools in T3.2 and WP5 tools Integrated Development Environment (IDE) in T3.3. The linking with WP4 comes from the elaboration of communication primitives (T4.1), data locality primitives (T4.2), security of AI/ML services (T4.3) and rules for verification of AI/ML services (T4.4). The relation to WP6 can be defined through the AI/ML models in T6.1, the data management and replication primitives in T6.2 and the runtime monitoring and reconfiguration tools in T6.3. The connection between WP5 and WP7 can be identified by Minimum Viable Product (MVP) and gaps and refinements to WP5 tools (T7.1), theoretical validation of WP5 toolbox within TaRDIS (T7.2), validation of WP5 toolbox within the use cases (T7.3), integration of WP5 tools (T7.4) and evaluation and benchmarking of WP5 tools (T7.5). Finally, WP8 receives input from and interacts with all WPs, and therefore, it is also linked with WP5.

1.2 Results Summary

First, the advances regarding Task 5.1, which aims at the development of AI/ML primitives are presented. Two important directions can be identified in this context. One of them is the introduction and description of the PTB-FLA framework, meant for implementation of federated learning (FL) algorithms. The development paradigm is described, an example of implementation is provided and possible TaRDIS usages are identified. The second direction is the development of algorithms in the Flower framework. Besides the description of developed implementations and their corresponding evaluations, ideas for further algorithm development are identified with possible positioning in TaRDIS use cases. Further, the advances in Task 5.2, oriented towards AI-driven planning, deployment and orchestration framework are explained in this report. The simulator for computer networks and the environment used for simulation are described first. Furthermore, the centralized RL agents that use the simulator are introduced and an explanation of the implemented DDQN algorithm is provided. Finally, some clear ideas are given regarding the further steps for the upcoming period. The advances in Task 5.3, regarding the development of a library of lightweight and energy efficient ML techniques are also discussed in this document. Three techniques for making a ML model more lightweight and energy-efficient are presented: early

exit of inference, knowledge distillation and pruning. The pruning method, identified as the most critical, has been developed. All these techniques are explained in detail, including different aspects, limitations, as well as input and output parameters and their tailoring to TaRDIS use cases. The positioning of these techniques within the use cases is also included as well as the planned advances for the next period. Besides the detailed description of the technical advances on different tasks of WP5, the ML modeling of the TaRDIS use cases is also provided, for all TaRDIS use cases.

1.3 Deliverable Structure

The structure of this document is as follows. First, we provide an introduction that highlights the main ideas and results that represent the core of this deliverable (Section 1). We then describe the positioning of ML/AI tools in the overall TaRDIS framework (Section 2). Section 3 is dedicated to describing the ML modeling approaches for the TaRDIS use cases. In Section 4, we discuss the advances regarding Task 5.1. Subsequently, in Section 5, we represent the advances related to Task 5.2. The advances regarding Task 5.3, are presented in Section 6. Finally, we conclude this report by summarizing the main outcomes and highlighting the next steps in Section 7.

Funded by the European Union

2 Positioning of ML/AI tools in the TaRDIS framework

This section is dedicated to demonstrating the relation of WP5 with other technical activities within the TaRDIS toolbox (WP3, WP4 and WP6), as well as with the requirements analysis (WP2). The relation with TaRDIS use cases (WP7) is described in Section 3.

2.1 Overview and relation with TaRDIS requirements

The objective of WP2 is to analyze and review the end-user needs and determine the functional requirements of the TaRDIS development environment. Deliverable D2.2 [1] synthesizes the functional requirements for TaRDIS. It includes the generic use case requirements, the toolbox requirements and the use case requirements. The generic use case implies a generic business or project that intends to use the swarm development paradigm. It defines the common building elements that should be applicable to that project. The requirements regarding this generic use case are meant to be applicable to different business cases. The use case and toolbox requirements are more specific. Moreover, the particular use case requirements per use case provider of the TaRDIS project (i.e., EDP, TID, GMV, ACT) help determine the toolbox requirements that are divided into functional and non-functional ones.

WP5 contributed to Deliverable D2.2 [1] by identifying the requirements regarding WP5. The requirement *"A list of provided FL and RL algorithms, including also unsupervised methods, with a data preprocessing and pre-trained models and incremental model retrain facilities, as well as ML inference and evaluation"* was added to the list of generic use case requirement by WP5. It consolidates the generic form of toolbox requirements. This incorporates the work from the various tasks within WP5, described in this report. WP5 added 9 functional toolbox requirements that are closely related to the corresponding use case requirements. The toolbox requirements named *"A list of provided FL algorithms", "A support for incremental model retraining within FL algorithms", "A data preprocessing facility for FL"* and "*A support for ML inference and evaluation"* are aligned with the content of Section 4, where the advances in T5.1, regarding FL algorithms development are described. Further, the requirements named *"A simulation environment for training of RL agents", "Centralized RL agent for task offloading"* and "*Decentralized RL agent for task offloading"* are aligned within Task 5.2, and can be understood through the concepts and advances given in Section 5. Finally, the toolbox requirements *"Support diverse ML algorithms in decentralized frameworks"* and *"Lightweight techniques for ML training and inference"* are aligned with T5.3, through the detailed explanation of techniques and advances in Section 6. Moreover, these requirements and their relations to the use case requirements are also present in the descriptions of the use cases and the corresponding ML models in Section 3.

2.2 Interaction with TaRDIS programming abstractions

The goal of TaRDIS WP3 is to specify the programming model and the APIs (Application Programmer Interfaces) that will be offered to programmers by the TaRDIS toolbox. Moreover, WP3 will develop an IDE (Integrated Development Environment) that will help programmers in writing software based on the TaRDIS toolbox. In essence, the TaRDIS APIs and IDE will be the entry points allowing programmers to develop reliable heterogeneous swarm applications by accessing the results developed by TaRDIS WP4, WP5, and WP6. A first outline of the TaRDIS programming model and APIs is provided in Deliverable D3.1 [2] (led by WP3).

Funded by the European Union

At this stage of the project, WP5 has contributed to Deliverable D3.1 [2] and WP3 by providing initial outlines of the APIs that WP5 will develop throughout the project, and that will be included in the TaRDIS toolbox. Such APIs are based on the TaRDIS use case requirements, and they will allow programmers to access the distributed AI and AI-based orchestration techniques developed by WP5; the API outlines cover AI/ML programming primitives (T5.1), AI-driven planning, deployment & orchestration (T5.2), and Lightweight and energy-efficient ML library (T5.3). For more details, please refer to Deliverable D3.1 [2]. The API outlines will be further refined during the TaRDIS project, converging towards the final version that will be included in the TaRDIS toolbox.

2.3 Interaction with TaRDIS property verification tools

The goal of TaRDIS WP4 is to develop novel formal analyses for determining whether a heterogeneous swarm is sound, secure, and reliable. Specifically, analyses will apply to the TaRDIS models to ensure that desirable security, data integrity, AI coordination, and communication properties are satisfied, with properties chosen according to the TaRDIS use cases and requirements. Moreover, WP4 will facilitate safe usage of the AI and data primitives from WP5 and WP6. The developed tools will be incorporated into the TaRDIS APIs and IDE and AI optimisation framework. A first outline of the list of identified desirable properties for which analyses will be designed in the rest of WP4 is provided in Deliverable D4.1 [3] (led by WP4); the properties have been categorized according to the task to which they have been assigned.

At this stage of the project, WP5 has contributed to Deliverable D4.1 [3] and WP4 by providing initial definitions of properties that WP5 will develop throughout the project, and that will be included in the TaRDIS toolbox. In particular, T4.4 that concerns the deployment and orchestration integration is strongly in line with WP5. T4.4 will facilitate the integration of the analyses developed in the areas of communication behavior (T4.1), data convergence and integrity (T4.2), and security (T4.3) with the techniques advanced in AI-driven planning, deployment, and orchestration (T5.2), enabling the automatic generation of optimized models that satisfy given desirable properties. For more details, please refer to Deliverable D4.1 [3]. The toolset for communication, data, AI/ML, and security analyses, outlining the developed analyses for desirable properties, will be further extended and refined, ultimately leading to the final version that will be integrated into the TaRDIS toolbox.

2.4 Interaction with TaRDIS data management and distribution primitives

WP6 is working on providing communication, membership and data management primitives.The relation between WP5 and WP6 comes from the aim to provide the ability to combine ML with decision making regarding resource availability.

Through the results achieved within WP6, the designed platform will be able to collect, aggregate, transform and store diverse monitoring-related data from heterogeneous sources and with respect to resource availability. Users will be able to have access to a unified interface of the system and application state progression over time, which will allow for informed decision making. If the decision process is to be automated, the platform will offer an API for machine learning models to be both trained on and put in the role of decision makers. The toolbox stores telemetry data (e.g. CPU load, disk usage, memory usage, network properties...) in time-series manner (i.e. data points with time). Machine learning models can access this data through the specifically designed API geared towards their needs. This simply means that any agent that requires data from the platform for training, needs to send a point in time when the last contact occurred. With this, machine learning

Funded by the European Union

models can be trained at their own pace, and agents cannot miss or skip newly added data points, even if an agent crashes, or goes offline for some period of time.

In the context of task 5.2, WP5 is responsible for automated analysis and planning, while WP6 will perform execution and monitoring, according to the framework MAPE-K [4].

Several ML algorithms need to be supported by the TaRDIS toolkit, in connection with WP6, including supervised learning algorithms (e.g., for regression and classification tasks, as well as for time-series forecasting), unsupervised learning algorithms (e.g., anomaly detection tasks) and reinforcement learning algorithms (e.g., decision-making for resource optimization). All these algorithms must be supported in their federated version during the training phase. The most crucial connection is the capture of network and node usage metrics to be fed to the orchestration RL agent (or agents), and the communication between agents in the deployment phase. These are detailed in the requirements Deliverable D2.2 [1].

Funded by the European Union

3 ML MODELING OF TARDIS USE CASES

This section provides initial ML modeling approaches for all 4 TaRDIS use cases, in accordance with Deliverable D2.2 [1].

3.1 EDP USE CASE

3.1.1. Introduction

The distributed swarm application for the energy exchange and balance will be utilized by Distributed System Operators (DSO), energy community managers, consumers, producers or prosumers within an energy community, etc. The general requirements that shall be fulfilled by the functionalities of the application are two-fold (see Deliverable D2.1 [5] and Deliverable D2.2 [1]): (i) energy nodes (consumers and producers) inside an energy community (10-20 households) shall be able to exchange plans of energy needs and local production in a forecasting period of time via a centralized entity, namely the energy community orchestrator; (ii) community orchestrators shall enable the energy exchange between different communities through horizontal communication frameworks, promoting the intra and inter--community energy balance.

In order to design the Machine Learning (ML) functionalities of a distributed application for energy exchange and balance, several objectives have to be taken into consideration:

- One of the primary goals of the system is to maximize the use of renewable energy inside a community. To this end, it is preferred that the energy that is generated locally (e.g., via solar panels or wind turbines) be consumed directly at the premises of the producer, or locally inside the energy community.
- All the consumers, within the energy community, should have direct access to energy supply that covers their immediate or scheduled request. The system shall therefore guarantee the energy supply of the consumers at all times.
- The energy application should reduce the number of messages between peers within the communication network. In the described framework, the nodes exchange messages via the centralized node (orchestrator) and the number of these messages should remain as low as possible, using the same coding and equipment for all actors.
- The system should enable the future incorporation of intraday market bid methods, allowing consumers and producers to trade (buy and/or sell) energy directly to/from the market.

In the design of the application for the energy use case, multiple assumptions and constraints also arise (see Deliverable D2.1 [5] and Deliverable D2.2 [1]):

- In case that the energy exchange among the energy community members does not suffice (considering the energy deficit or surplus of each individual smart home), the grid provides the missing energy, albeit the most expensive option. This should be considered as an “option of last resort”, since it is the highest order of energy providers.
- The energy community is actually a smart grid, consisting of several energy nodes (providers/producers and consumers) that can draw or provide energy. The neighboring communities can also exchange energy among them.
- The entity prosumer (i.e., energy consumer and producer at the same time) represents a community member (ie., a household or an electric vehicle charger - EV charger) that can be registered in the system/application for consuming energy only,

Funded by the European Union

or for consuming and producing energy. Producing energy in households assumes some energy source like photovoltaics (PV), micro wind turbine generators (mWTGs), but can also come from battery energy storage systems (BESS).

- Some EV Chargers can also act like producers, since they are able to provide energy that is locally stored (charged EV or local battery storage) back to the community power grid. Moreover, there are two modes of an EV charger: public (the EV charger is registered as a standalone prosumer) and private (the EV charger is connected to the grid and can be connected to a car to charge its battery).
- As aforementioned, a special logical entity in the design of the system is the Community Orchestrator, which is used for cross-community operations (offering surplus production from one community to the other, or requesting additional supply from other communities when local production does not meet demand).

3.1.2. Machine Learning functionalities involved in the energy use case

By following the internal logic of the decentralized application for the energy use case, the architectural view of the ML algorithms that will be included is shown in Figure 1. Evidently, each energy node exists in a smart home local environment, possibly generating some energy source like photovoltaics (PV), micro wind turbine generators (mWTGs), storing the energy (energy storage system, i.e., battery) and also exhibiting its individual energy consumption requests. In this context, the data that are present in the local environment can only be accessed by the local ML model of its own smart home. Moreover, the smart homes are also connected to an energy smart meter that registers the production and consumption requests and connects the households to the utility power grid. The community orchestrator entity is located at a higher architectural level and it is connected to all the smart homes, being able to exchange data (concerning mainly the individual energy consumption needs/requests and the renewable energy generation offers) with the community. A global model resides in the orchestrator, targeting to provide collaborative intelligence for each ML task that is designed for optimization of the smart home resources, also involving the higher level of observability (the orchestrator can gather data from all local smart home environments). The ML functionalities that will be designed for the energy use case can be described as follows:

- Time-series prediction/forecast. This ML model will enable the **prediction/ forecasting** of some of the smart homes' parameters in the next time instance (or in the next few time instances) based on historical data. The prediction should consider the individual time-varying parameters of the home, e.g. solar panel-produced power over time, battery level over time or even power consumption requests over time for each smart home. Most of these parameters are periodical or at least exhibit seasonality (for instance, the average solar panel generated power is expected to be lower in winter for the northern hemisphere). In any case, we will use past historical values of these parameters to forecast future values for each house that can be periodically acknowledged to the community orchestrator. For this objective, we will use Long Short Term Memory (LSTM) neural networks. It should be noted that the community orchestrator can also include additional matching functionalities for purposes of intra-community energy balance, i.e., matching algorithms between consumption and production requests that target the net neutral energy balance. Furthermore, the higher level of matching involves inter-community energy balance, where the community orchestrators coordinate amongst them in order to exchange (draw or provide) power and minimize the grid usage.
- Deep Reinforcement Learning (DRL) algorithm for **controlling the power** for the Heating, Ventilation and Air-Conditioning (HVAC) system inside the smart home. We

will use the output of the LSTMs (i.e., the predicted solar panel power, requested power for load activation, battery level, etc.) as input to the DRL algorithm for each individual environment. The DRL algorithm will be therefore tailored for each individual smart home and will decide (suggest/recommend actions) on whether to provide input power from the grid to the HVAC system or provide input power to battery for charging purposes. The state of each environment will be observed at each time point and the intelligent DRL agent will be the responsible entity for providing an action that is beneficial for the environment, according to the objectives set during the training process of the ML model. The state of the environment, includes among other variables, the temperature inside the smart home, the ambient temperature, the battery level, the consumption requests, the power gathered by the solar panels or other renewable energy sources, the energy cost, etc. It is worth mentioning that these variables for each individual smart home are time-varying. The objective here is to maintain a comfortable temperature range inside the home, while at the same time minimizing the energy cost (energy that is drawn from the grid should be minimal). This ML model also promotes the general objectives of this use case, since we target to maximize the use of renewable energy inside a smart home, covering its specific energy needs via local renewable energy, while also minimizing the grid utilization and the associated cost.

- The above two ML models are completely decentralized, since each smart home has its own individual models that are trained with their own historical (or simulated in the case of DRL) datasets and are inferred with the states that are observed from the individual environments (either past-values of the time series or the current environment state in the case of the DRL model). Hence, the ML models do not interact and do not share intelligence with the other smart homes in the community. The idea of the extended version of the ML models is to enable collaborative intelligence through the orchestrator, using federated learning (FL) methods to facilitate the training process. As depicted in Figure 1, the community orchestrator acts as a server in the FL framework, while the DRL agents of each smart home act as clients. In this way, the ML models share intelligence by fusing or aggregating the model parameters (for instance FedAvg method computes the average of all model parameters [6]). The resulting model efficiency and training time in the completely decentralized scheme (each home has its own models with no visibility/ communication with other homes) and the federated learning scheme (homes share intelligence during training) can be then compared. For the latter federation, additional methods can be explored instead of FedAvg, i.e. FedProx, etc, as well as transfer learning methods, i.e. a model that has been trained using the data from one smart home can be transferred to other homes without additional training. To this end, the model accuracy during the inference phase can be compared between a received ML model through transfer learning in cases that a new smart home enters the community and the federated model transfer that has been constructed using the collective intelligence of many ML models and associated datasets.

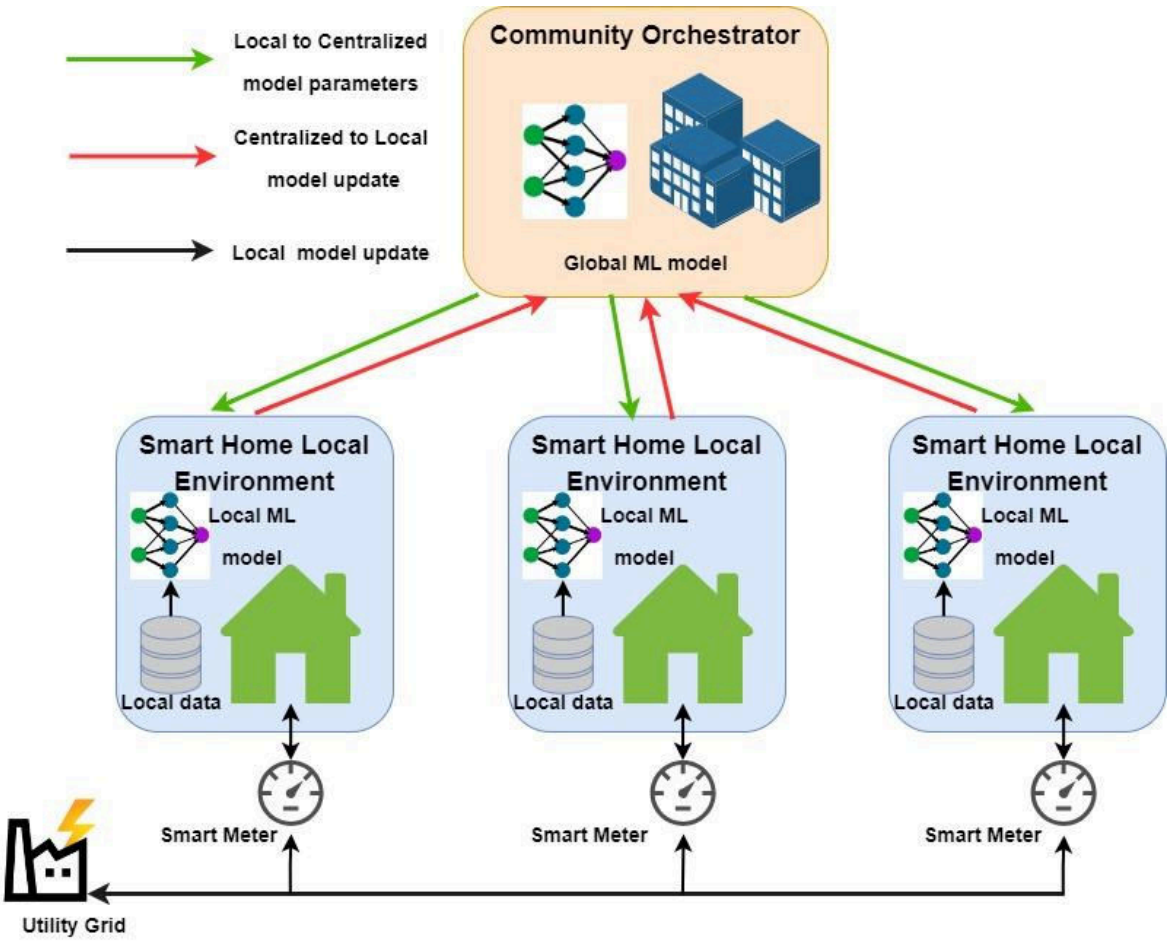

Figure 1: Architectural view of the designed ML framework and the involved entities in the energy use case.

3.1.3. Time-Series forecast with LSTM Model

The general principles of the LSTM neural networks that will be utilized in the energy use case are shown in Figure 2. The LSTM network takes as input a time-series of a variable (for instance the power gathered by the solar panels in a smart home environment), processes it and outputs the prediction values of this variable at future times instances [7]. More complex LSTM networks can also be considered, taking as input time-series of multiple variables. It should be noted, however, that the input variables should not be in general correlated. Furthermore, two parameters are used to regulate the training and forecasting capabilities of the LSTM network, namely the lookback window N and the forecasting moment M. The former reflects the time window needed to make an accurate prediction (or the number of past-values that have to be considered for an adequate forecast) and the latter reflects how far ahead in time the model makes the prediction (e.g., at the next time instance for $M = 1$).

Figure 2: General consideration of the LSTM network.

An LSTM model was trained with a dataset containing power values gathered from solar panels in a smart home in Denmark Copenhagen area [8]. The dataset includes: (i) hourly outside (ambient) temperature (Celsius); (ii) hourly power generation from solar panel

Funded by the European Union

(Watt/m^2); (iii) hourly energy price (Euro/KWh). These variables have been registered for the duration of 1 year (and specifically during 2014). No power consumption requests from the smart home are available. In Figure 3, the temporal variations of the power generated from the solar panel is illustrated.

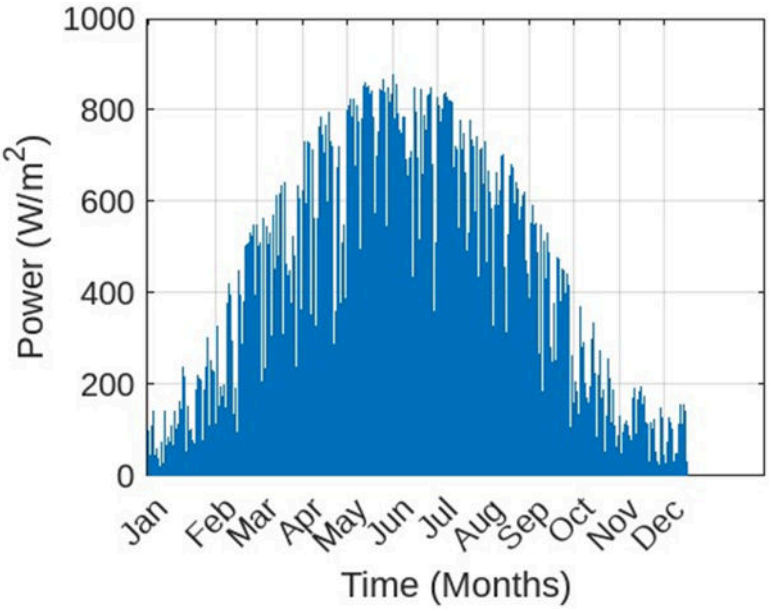

Figure 3: Time-series of the power generated from the solar panel in the area of Copenhagen during the months of January to December 2014.

The power time-series values generated from the solar panel were therefore used to train the LSTM neural network. Specifically, the training data were considered the hourly power gathered by the panels during the months of January to November, whereas the power gathered during the month December was used as testing data (LSTM did not encounter the testing data during the training process). The resulting fine-tuned (learning rate, optimizer, lookback window) LSTM model predictions are depicted in Figure 4, achieving a mean error of approximately 7.899 W/m^2. Although the model achieves adequate accuracy predictions, following the time-domain trends, additional data are required from several years to achieve higher accuracy. In particular, the LSTM model has not been trained with solar power data from month December of other years that exhibit daily periodicity and yearly seasonality (the solar power data from month December was left out as the validation dataset and was not used during the training phase of the model).

Funded by the European Union

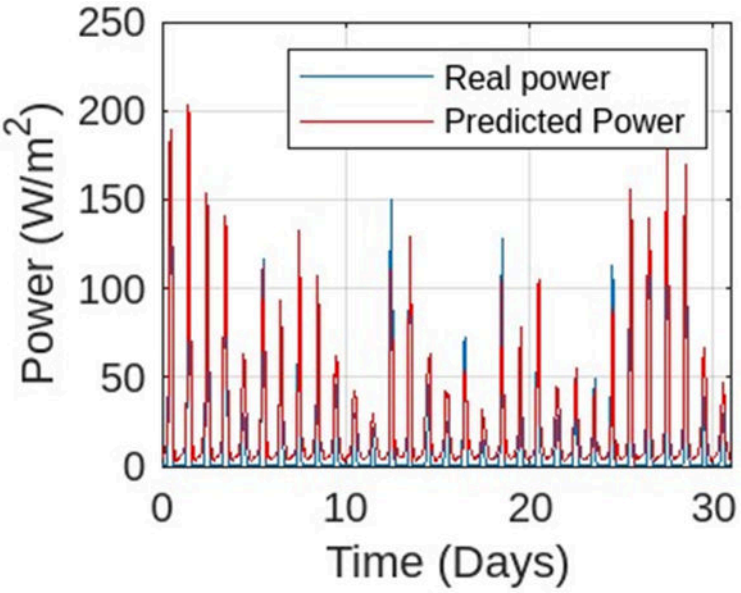

Figure 4: Real vs Predicted solar panel power for the testing data (December 2014).

3.1.4. Home Energy Management System with DRL Model

In this section, the outline of the Home Energy Management System (HEMS) algorithm is provided. We assume that the smart home consists of (as shown in Figure 5):

- Distributed generators, that can be solar panels or wind generators and are used to gather or collect power from renewable sources
- Energy Storage System (ESS) that can be used to store energy (for instance batteries), and can be utilized to reduce the net-energy demand from the grids and provide energy that is stored locally
- Loads inside the smart home that can be
    - Non-shiftable loads that need to be satisfied immediately (e.g., televisions, microwaves, and computers).
    - Shiftable and noni-nterruptible loads (for example washing machines). Their operation can be scheduled but cannot be interrupted.
    - Controllable loads (e.g., the HVAC system, heat pumps, etc.). These power consumption requests can be controlled by an intelligence entity (DRL agent).
- The HEMS that operates in discrete time slots $t \in [1, T]$, where T is the total number of time slots. We assume that in each time slot the HEMS makes a two-fold decision: (i) decides upon the ESS charging/ discharging power and (ii) decides upon the HVAC input power. The aforementioned decision is made based on the rest of the variables that are observed by the smart home environment, i.e., ESS level, power consumption requests, etc.

The target of the HEMS agent is to regulate the provided power to the HVAC system and the ESS charging/ discharging power in each time slot in order to maintain the temperature comfort inside the smart home, as well as minimize the energy cost provided by the grid, therefore inherently maximizing the use of renewable energy.

It is known from literature that the Deep Deterministic Policy Gradient (DDPG) design can tackle problems that require continuous action space and belongs in the actor/critic methods which are off-policy methods [5]. This means that they use two independent components, the actor network and critic network, to update the policy and value function. The complete mathematical formulation of the HEMS algorithm can be found in Annex 1. The actor network

takes as input the state space s_t and delivers the desired actions a_t. Then both the environment s_t and actions a_t are being fed as inputs to the critic network which outputs a Q-value function $Q(s_t, a_t)$. Using experience replay and target networks, the critic network updates its weights and the actor network updates its weights by computing the policy gradient. After that we softly update the target networks, to ensure a stable training process and repeat this algorithm until convergence. The DRL (DDPG) agent and the learning process (including the observability of the environment and the interaction with the smart home) are depicted in Figure 5.

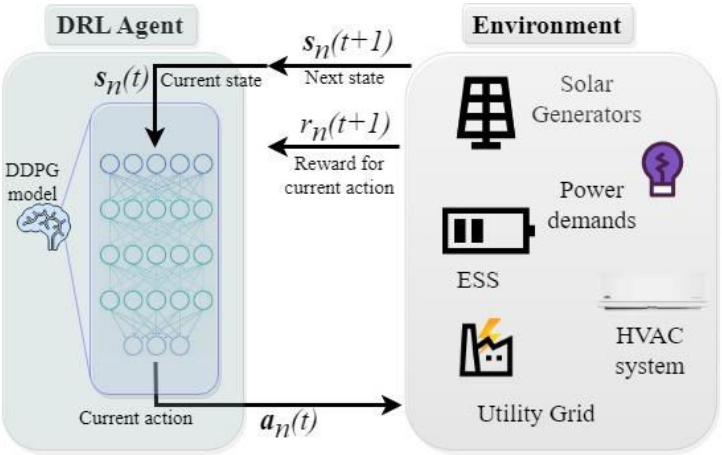

Figure 5: Representation of the interaction between the HEMS DRL Agent and the Smart Home Environment.

In the preliminary results presented below, we used two hidden layers in the actor network and four hidden layers in the critic network. The same dataset (from the Copenhagen area) was used to train the DDPG model only for temperature comfort. The training data were again selected from November to December and each training episode was selected to range: (i) random selection of 1 day (24 hours); (ii) random selection of 1 week (24*7 hours); (iii) random selection of 1 month (24*31 hours). In specific, these are the steps within each episode that the agent can interact with the environment, taking into account the temperature, the ESS, the renewable energy, the pricing and the building dynamics. In the next episode the environment is randomly initialized and the intelligent agent starts again by experimenting with the environment. It should be noted that a ε-decay policy was selected for the transition between the exploration and exploitation phases, while the comfort disutility reward reaches a maximum value of 0 if $19.5 \leq T_t \leq 22$.

The DDPG models are training with different values of learning rate, concluding that the learning rate of 0.001 seems appropriate as it demonstrates convergence to zero value of the rewarding function after 1500 iterations and exhibits low fluctuations with variability during the explore-exploit phase. Then, with the chosen learning rate of 0.001 we proceed by training with different steps [1 day, 1 week, 1 month] within each episode to explore the influence it has on model convergence. In Figure 6 we depict the learning curve with a varying number of steps per episode. Based on the results, we conclude that there is no significant difference between convergence across time periods. Nevertheless, we choose the time period of 1 day for two primary reasons: first, we want our model to train faster in order to adapt to the changing conditions of the environment and user needs and, second, due to its ease of implementation and cost-effectiveness.

Funded by the European Union

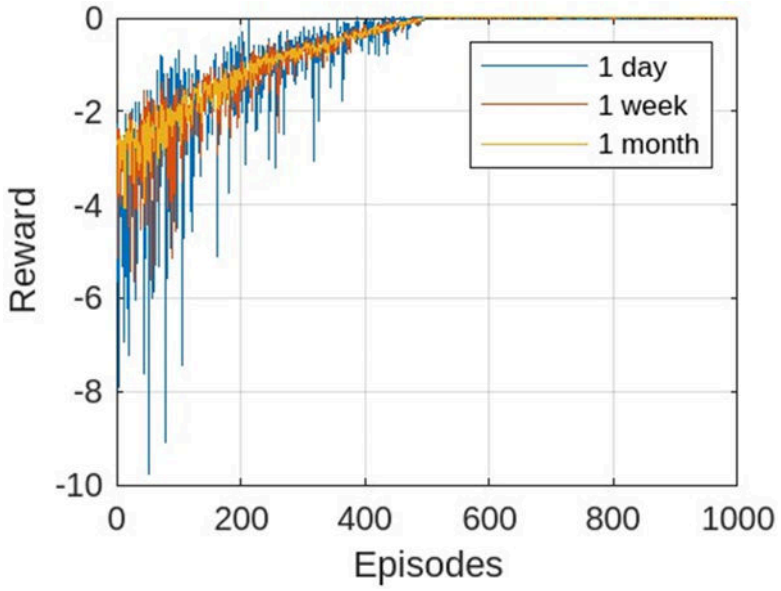

Figure 6: Learning curves for different numbers of steps within each training episode.

Having trained the DDPG models, we can infer them using the testing set (December). The inference action on the HVAC system for 1 random day of December (24 hourly steps) is shown in Figure 7 (left panel), while the related inside temperature is shown in Figure 7 (right panel). Evidently, the DDPG agent stabilizes the time-domain temperature regardless of the ambient temperature and solar panel produced power by providing the required input power to the HVAC to keep the temperature within the comfort bounds.

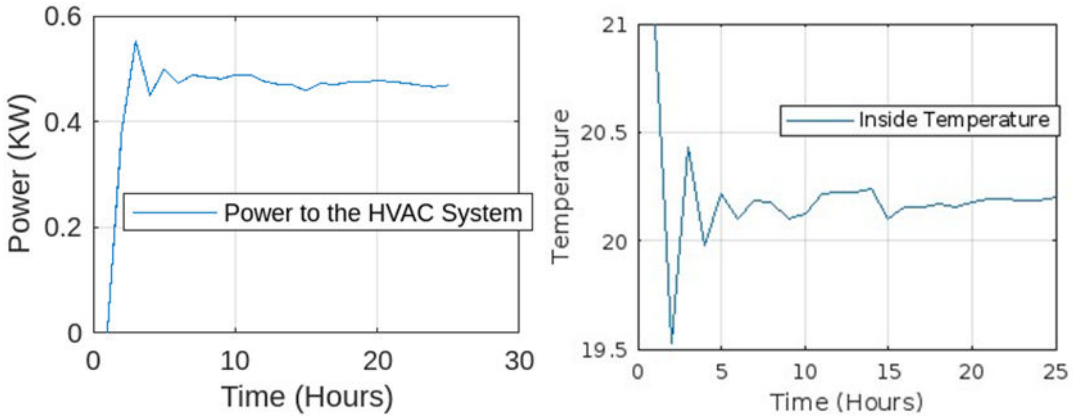

Figure 7: Power provided to the HVAC system during the inference of the pre-trained DDPG model (Left Panel). Temperature variation inside the smart home that is controlled by the HEMS agent (Right Panel).

3.1.5. Extensions in the Tardis project lifetime

The following extensions will be implemented in the aforementioned algorithms:

- The DRL state will take into account the renewable generation output r_{t+1} and the non-shiftable power demand g_{t+1} at future time instances (see Annex 1 for the state space). In particular, the forecasting capabilities of the LSTM models will be used to feed the state space of the DDPG model, making the DRL algorithm more proactive.
- Moreover, the DDPG model will become more sophisticated, including multiple conflicting objectives. Specifically, the reward function may include the concept of

selling the generated energy back to the grid, targeting the ultimate target of inter-community energy balance.
- The DDPG model has been tested for a single smart house with acceptable results. In the upcoming period, the DRL-Federated algorithm will be developed, incorporating the learning process of multiple smart houses and collaborative intelligence.
- Datasets (time-series) for verification of the federated learning process from multiple smart homes have already been identified, including historical data from realistic energy consumption requests [14].

3.2 TID USE CASE

TID's use case concerns intelligent homes where different devices are part of an automated system that is designed to facilitate everyday life. In particular, this automated system monitors and/or controls home attributes such as lighting, climate, entertainment systems, and appliances. Typical examples of such devices are personal assistant devices, smart TVs, mobile phones, smart light switches, etc. Through federated learning (FL), these devices collaboratively train deep neural networks on their local datasets (samples and labels).

Given the heterogeneity of available resources (computation, energy, memory) of the aforementioned devices, use case scenarios that TID is interested in include the presence of hierarchy in such systems as well as the split learning (SL) paradigm. The former can exploit the hierarchy that sometimes naturally occurs (e.g., compute-enabled routers that may be higher in the hierarchy than any other home device) for FL-related processes such as aggregation or ensuring privacy. The latter allows part of the model training to be offloaded as a processing task to a device (nearby) with available resources. These scenarios capture the characteristics of a smart home since the devices that are part of a smart home system may be resource-constrained (and, thus, SL allows them to participate in the training) and also may be vulnerable to privacy concerns (and, thus, the hierarchy in the system may address this issue by ensuring that the trusted devices insert noise to the data shared with the FL aggregator).

Possible use case applications and related data sets and models that TID envisions to implement (the applications were mentioned also in Deliverable 2.2 [1]):

1. image classification (using state-of-the-art models such as MobileNetV2 [15] and above, and datasets such as CIFAR10 or MNIST, etc.)
2. natural language processing problems such as text modeling for sentiment prediction that predicts the sentiment of movie reviews as either positive or negative (with state-of-the-art data and models such as the IMDB dataset [16] and convolutional NN or long short-term memory (LSTM) NN)
3. predicting when to deliver notifications for mobile applications [17] (with state-of-the art models and open-sourced data)
4. wake-up word speech recognition by a device when it is pronounced by the user [18]
5. natural language processing problems such as next-word-prediction for the keyboard [19].

These applications above capture processes that may be in place in intelligent homes. For example, next-word prediction could be used for the search bar in an application. Moreover, the text modeling for sentiment prediction is typically used to analyze the user's feedback and obtain insights into what the user will like in the future, for example in the recommendation system of an on-demand video streaming service.

TID will in principle make use of open-source data sets, while at the same time exploring the possibility of having access to a dataset from Movistar Home (the product of TID). We plan to simulate or emulate such application scenarios in Android-based devices (eg. mobile phones, Android-based personal assistant devices).

As a baseline implementation, TID has conceived and developed the federated learning-as-a-service (FLaaS) middleware that allows a cross-application and cross-device federated learning. APIs and programming primitives that are part of the TaRDIS toolbox are expected to be incorporated in or work along this middleware to improve on the expected KPIs. Specifically, FLaaS may benefit from the output of tasks 5.1 and 5.3, upon compatibility. For example, FLaaS is developed [20] using TensorFlow Lite library that allows the developer to orchestrate the training in Android-based devices. A potential development of an improved library (that incorporated functionalities that are beneficial for the use case) could be easily incorporated in the current version of FLaaS. Moreover, TID might consider applying different aggregation functions beyond the typical FedAvg like the ones developed within task 5.1. See for example Section 4.2 here. Finally, methods ensuring energy efficiency can be very valuable in TID's use case as devices may need to participate in a ML training or an inferring task while limiting the corresponding energy consumption.

3.3 GMV USE CASE

The GMV use case aims at achieving on-board distributed autonomous Orbit Determination and Time Synchronization (ODTS) for a large constellation of satellites in Low Earth Orbit (LEO). A constellation cycle represents the time after which the ground-track repeats. This means that the positions of the satellites are the same respect to the Earth Centered Earth Fixed reference frame. The constellation taken as a reference is characterized by 170 satellites, where the orbital period of each satellite is 1.8 hours, and one whole constellation cycle is 7 days. The satellites communicate by Inter-Satellite-Link (ISL), which is also used for ranging measurements, and four ground stations are available for additional measurements and communication with Earth.

Orbit Determination and Time Synchronization (ODTS) in the GMV use case is achieved by means of an Extended Kalman Filter (EKF). The EKF, based on a dynamical and an observation model, depending on sensor measurements and their reliability, provides the estimate that minimizes the satellite state vector covariance.This is common practice in space engineering when dealing with the non-linear problem of ODTS. However, interesting research has been going on recently with respect to the application of Machine Learning algorithms to both orbit propagation and estimation with promising results [21].

One of the fundamental aspects of the orbit determination algorithm design is that it needs to be developed according to the real world scenario properties. In particular, the measurements used for the algorithm testing shall be as realistic as possible. In space engineering applications, in fact, simulations play a huge role, since in-orbit direct testing of critical functionalities is definitely not an option.

GMV in-house knowledge will be used to develop a realistic noise model, based on real measurements and state of the art literature review. The intention is to simulate radio frequency (RF) pseudorange/carrier phase and Doppler measurements replicating the behavior of a Global Navigation Satellite System-like (GNSS-like) receiver. An optical measurements model might also be taken into consideration. GMV has an in-house optical Inter Satellite Link (ISL) simulator that could be used for this purpose.

The candidates under test for the decentralized ODTS algorithm developed for the GMV use case are a set of non-linear algorithms:

- **Gaussian Mixture Model [22]**
  A Gaussian Mixture model is a probabilistic model of a given random variable using a parameterized set of Gaussians, which can be used to describe virtually any probability distribution function given the proper parameters. The parameters of the model can be estimated using a number of algorithms, such as the EM (Expectation-Maximization [23]) , Variational Inference [24], the Gaussian Sum Filter [25] or even a Neural Network [26].

- **Neural Networks [27]**
  Neural Network is an umbrella term for a group of deep learning (DL) models, where each neuron performs a non-linear transformation of its input. In practice, there are many types of Neural Networks (Convolutional Neural Networks - CNN [28]; AutoEncoders; Transformers), each with its own use and specific architecture tailored to solve different problems. In this case, the orbit estimation task lends itself to a time-series task, using prior states and observations to obtain a precise current or future state, so it is necessary to use an architecture that is aware of the temporal relation between input data [29] [30].

- **Physics Informed Neural Networks [31]**
  Physics Informed Neural Networks (PINN) [32] are a subset of neural networks that combine the data-driven approach of Neural Networks, with known information on the dynamical system in which the estimation is being made[33]. This information can be incorporated into a Neural Network as a regularization term in the evaluated loss, as a knowledge boundary on the set of possible solutions, or as an initial solution [34] to be refined by the data-driven model.

They will be tested in the mentioned order, following the concept of starting with the simplest approach first and following with more complex options. A trade-off based on the resulting accuracy and computation time shall be carried out to determine the method of choice.

Referring to the Mock-up block diagram (Figure 8, also presented in Deliverable D7.1 [35]), which represents the structure of the ODTS simulator developed in Matlab/Simulink as a baseline, the target is to replace the entire EKF block with an ML model.

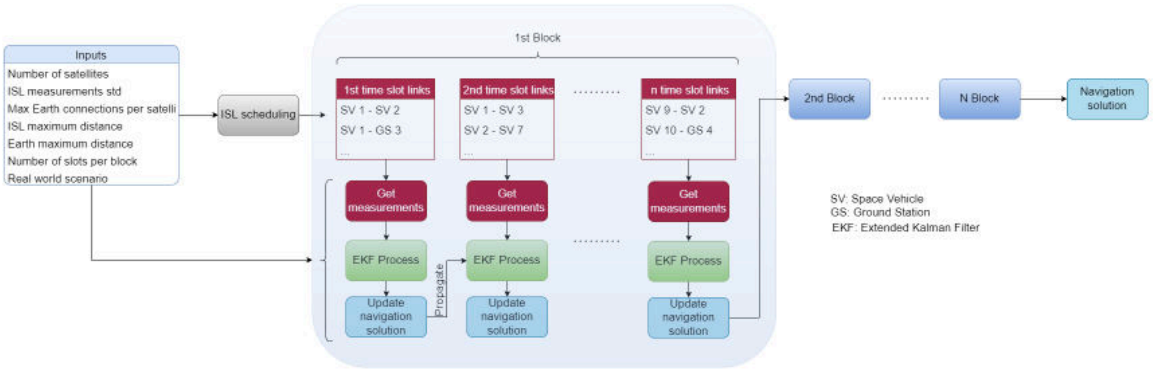

Figure 8: Mock-up block diagram

The ML model would take the inter-satellite and satellite to ground stations measurements as inputs and provide the updated satellites state vectors as outputs.

Moreover, another task would consist of developing an orbit propagator by means of an ML model to be used when no measurements are available. There are multiple advantages:

- Faster processing
- More accuracy
- Less heavy in terms of computational time
- Less parameters needed

An example of a very high level pseudocode of a sequential simulation of the distributed ODTS is outlined below (the pseudocode was defined in Deliverable D3.1 [2]).

```
settings = setParameters() // establish parameters settings for ODTS simulation
orbital_data = getOrbitalData(orbit_data_file) // get satellites orbital data
connections = islScheduling(orbital_data) // inter-sat connections over time
for block_number = 1 : N // iterate over blocks
      for t_slot = 1 : n // iterate over time slots
            for sv_number = 1 : n_sats
                  obs = getMeas(connections(t_slot),
                              orbital_data,
                              settings) // simulate measurements
                  state_update = odts(state, obs) // performing ODTS
                  state = state_update
            end
      end
end
```

TaRDIS APIs related to ML will be used within the 'odts' function mentioned in the pseudo-code. This work is to be developed by GMV and NOVA.

3.4 ACT USE CASE

This use case concerns a real smart factory, implemented by an ACT customer in collaboration with ACT. The main idea is the automation of the intralogistics of production lines for the assembly of machining centers. The intralogistics is responsible for moving all parts and materials between workstations and warehouses as required. The requirements regarding this use case are oriented towards factory automation, where heterogeneous intelligent swarms can be used to orchestrate the production process on the factory shop floor. There are multiple concerns present here: the resilience of the system, a common language and representation that ensures that the implementation of the logistics processes is faithful to their design, and the observation of the execution of logistic processes, preferably with an automatic classification into nominal and anomalous ones.

There are several tasks that could be of interest regarding ML: classification of workflows, device health status prediction, data placement via reinforcement learning and predictions under noisy labels. The focus here is on the anomaly detection of workflows, based on noisy labels.

The task is to perform anomaly detection of workflows, by means of training ML models for anomaly classification on both ongoing and past workflows. This implies development of models that accept both "full" and "partial" workflow as an input. This approach requires incremental training, where the ML model is refined on a small number of traces, in order to make it applicable to a concrete factory workflow, without having done extensive work (on real or simulated data, as available) for that particular process. There are no human

generated labels available for the training phase. Instead, "pseudo-labels" are obtainable at the time of data acquisition and for training. Human generated labels may be available for model accuracy evaluation. The positioning of ML models within the ACT use case can be understood by observing the ACT pseudocode (The manager dashboard application, as described in Deliverable D3.1 [2]):

begin:

query list of IDs of all maintenance tasks that were recently closed (⇒ query the event store)
remove task IDs that are already on closedTasks from this list
run ML inference to get nominal/anomalous status for each task's whole history (⇒ use ML API)
add heuristic labels to tasks (noisy) based on production expert inputs
append task IDs to closedTasks (to prevent them from being processed again)
feed labeled tasks into federated learning service (⇒ use ML API)

query list of IDs of all open maintenance tasks (⇒ query the event store)
run ML inference to get nominal/anomalous status for each task's whole history (⇒ use ML API)
compute current workflow state for each task (⇒ instantiate state machine)
replace openTasks with list of tuples (task ID, anomaly status, workflow state)

display closedTasks, openTasks, and FL training status & metrics in suitable UI, allowing manager to perform state updates like a worker (⇒ run machine command)

loop begin (after suitable delay)

The FL ML pseudocode can be viewed as follows:

Begin:
Load data and specify input parameters
Perform preprocessing
Set up the model
Begin loop:
Re-train the model with FL
Perform inference

A state of the art analysis for federated learning with noisy labels is ongoing, and it turns out that open data may be used to benchmark for FL under noisy labels. The idea for standardized benchmark, that enables federated noisy settings exploration is presented in a recent work [36]. The benchmark provides a comprehensive pipeline for generating noisy labels and heterogeneous FL settings, while supporting a set of 20 different federated scenes (these scenes enable a wide scope of data partition approaches and label noise generation mechanisms). This may be considered as an additional reference, in addition to the ACT specific data.

We propose an initial ML model for the ACT use case. The initial ML model is based on the requirements, previous discussions and related prior work. One of the main approaches we rely on, is presented in [37]. It also uses unsupervised learning regarding the nature of the problem, where no ground truth and labels are available. It proposes a hybrid unsupervised outlier detection method that combines the approaches of clustering and representational learning. Also, an important reference that motivates our work is integrating deep learning-based anomaly detection as a service into mobile cellular Internet of Things (IoT) architecture [38]. The approach embeds autoencoder based anomaly detection modules at the IoT devices, as well as in the mobile core network.

The proposed ML model for the ACT use case is presented in Figure 9. The model represents a highly flexible approach, as there is no need for human-generated labels, and it accounts for pseudo-labels. The model incorporates domain knowledge for pseudo-labeling and weighting features for clustering. Additionally, it performs internal anomaly validation and explanations, if required. Incremental training is allowed, as well as accepting partial workflow inputs.

Funded by the European Union

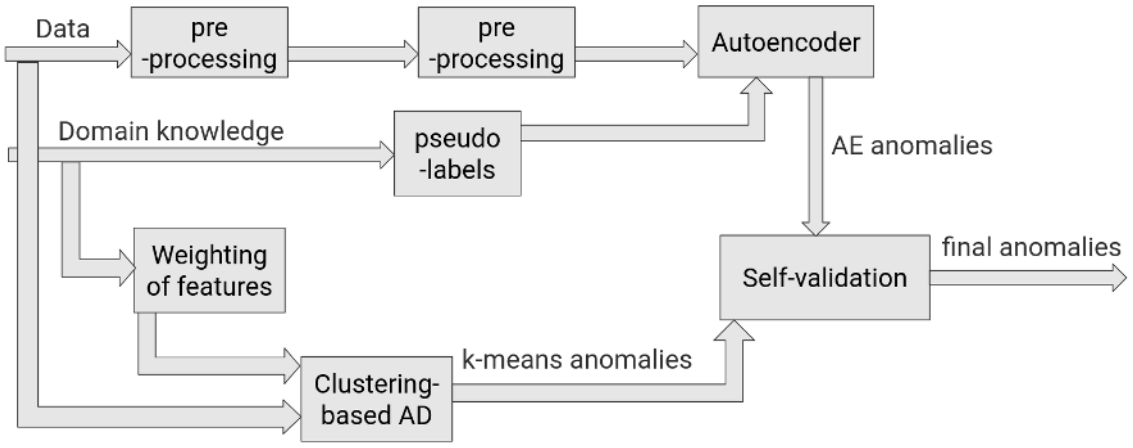

Figure 9: The overall approach for the ML model for the ACT use case

The training of the autoencoder is shown in Figure 10, while Figure 11 displays the autoencoder inference. The autoencoder is a neural network trained via stochastic gradient descent, or its accelerated versions like ADAM. Obviously, the incremental training of the autoencoder is achieved by default, as it allows naturally to be trained by incrementally adding new data points or data mini-batches. Regarding the issue of accepting "partial workflows" as model inputs for inference, it is non-trivial for autoencoders, but there are some existing solutions [39].

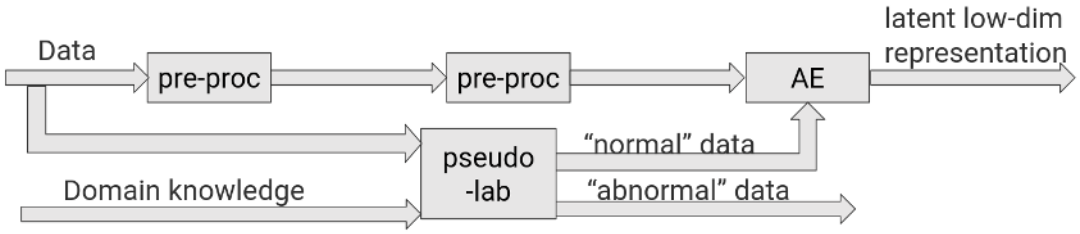

Figure 10: The autoencoder training

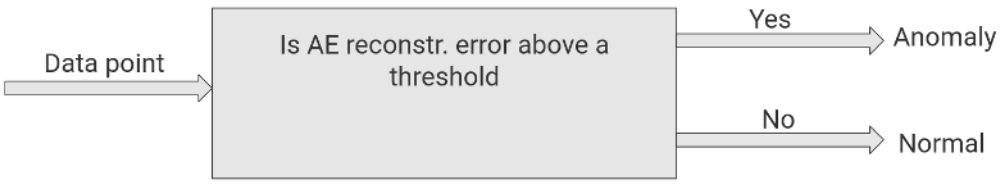

Figure 11: The autoencoder inference

Figure 12 shows the clustering procedure, while Figure 13 displays the inference. Providing the input data, the domain knowledge and the expected number of clusters (for k-means), the algorithm identifies singleton or low-cardinality clusters as anomalies. As an alternative to k-means, a convex clustering approach can be used, where there is no need to specify the number of clusters in advance. One approach for convex clustering that could be used was introduced recently [40]. It proposes a parallel Alternating Direction Method of Multipliers-based (ADMM-based) approach for clustering. Alternatively, a recent work [41] that introduces gradient based clustering as a general approach for distance based clustering, could be a useful solution as well.

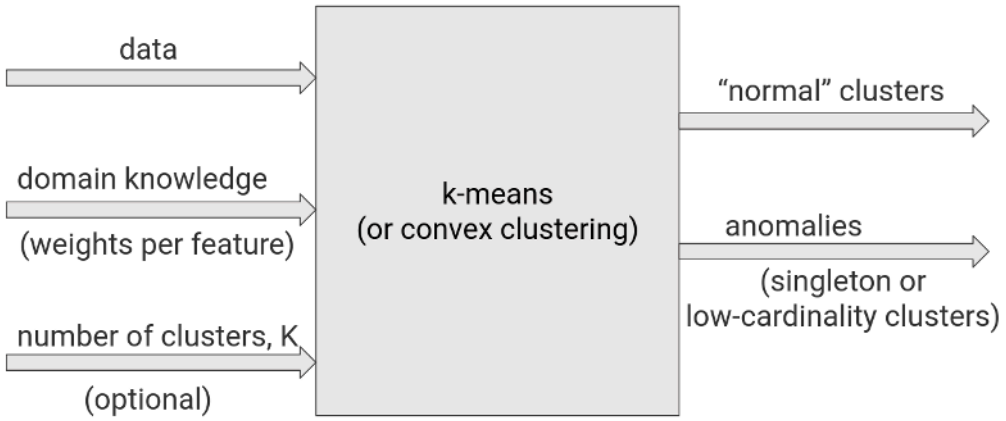

Figure 12: k-means training

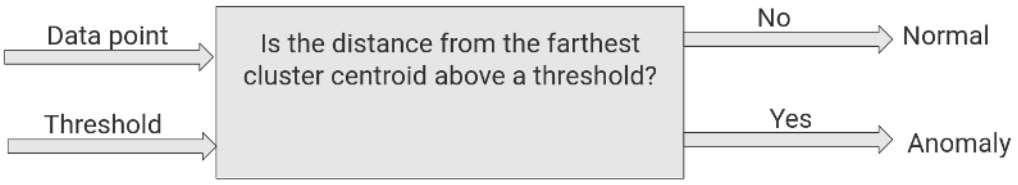

Figure 13: k-means inference

Enabling incremental training for k-means clustering is a non-trivial task, but there are some proposed solutions [42]. Even if a brute-force re-training of cluster centroids is done, if we assume a "forgetting factor" that only N data points in the past are relevant, the cost per iteration of retraining is linear in the number of data points N being re-considered. Regarding the acceptance of "partial workflows" as model inputs for inference, it works naturally for k-means clustering. Effectively, an inference point that has been partially observed will be assigned to the cluster that resembles most this data point based on the features observed (or is declared as an anomaly if no such cluster exists).

Considering federated learning, the proposed model currently assumes that all workflows (data) are available at a central server. If we assume that multiple groups of data points are available at multiple units (FL clients), and there is no "model partitioning" (i.e., all clients require a similar type of model), then both the autoencoder training and k-means training can be achieved in a federated/distributed setting.

4 Advances on framework supporting AI/ML modeling primitives

In this section, we describe two complementary frameworks, Python Testbed for Federated Learning Algorithms (PTB-FLA) and Flower, that are both of interest for use within the TaRDIS toolbox.

4.1 PTB-FLA federated learning testbed

Recently, Python Testbed for Federated Learning Algorithms (PTB-FLA) [43] was offered as a framework for developing federated learning algorithms (FLAs) i.e., as a runtime environment for FLAs under development, on a single computer (and on edge systems in the future). PTB-FLA was written in pure Python to keep application footprint small (to fit to IoTs) and to keep its installation simple (with no external dependencies).

We are living in a world where there are a lot of open-source software implementations of various ML/AI learning algorithms that are freely available in various online repositories, such as Microsoft GitHub, Google Colab, etc. But how can software developers reuse this software to develop FLAs for applications like TaRDIS use cases? The ad hoc approach may work well for simple algorithms, but for more complex algorithms some development paradigm is needed.

To that end, paper [44] presents the FLAs development paradigm based on PTB-FLA, which is used by software developers as a systematic guideline to transform a given sequential source code into the *semantically equivalent* (or *correct*) target PTB-FLA code (producing the same result as the source code). The main goals of PTB-FLA development paradigm are: (1) to aid the development of FLAs that are *correct by construction* (i.e., the paradigm guarantees target code correctness) and (2) to make the development of FLAs *easier* (i.e., easier than at hoc approach).

Next subsections present the development paradigm, the example of implementation – logistic regression, and the ongoing applications within TaRDIS use cases, respectively.

4.1.1 Development Paradigm

The PTB-FLA development paradigm is primarily intended to serve as a FLA developer guide through the process of developing a target FLA using PTB-FLA, which we call the FLA development process. The input to this process is the Python sequential program code of the target AI/ML algorithm, whereas the output from this process is the PTB-FLA code with the same semantics, which means that for given input data it produces the same output data with some tolerance *e*. The tolerance *e* is typically some small error value (ideally zero).

Of course, the output PTB-FLA code must be compliant with the PTB-FLA programming model which is a restricted programming model that imposes the following two restrictions: (1) using the Single Program Multiple Data (SPMD) pattern, and (2) specifying code for server and client roles in form of callback functions.

The main idea of the PTB-FLA development paradigm was to follow the principle of correct-by-construction, which when applied in this context meant to define the development process that would for a given referent code yield the output PTB-FLA code with the equivalent semantics. Following the approach used by program compilers, we devised the PTB-FLA development paradigm as the series of program code transformation phases

where each transformation phase consumes its input code and produces the semantically equivalent code that is closer to the target PTB-FLA code.

4.1.1.1 Development Phases

There are altogether four development phases, called phase 1, phase 2, phase 3, and phase 4, which are by the convention named by their output code i.e., (1) the referent sequential code, (2) the federated sequential code, (3) the federated sequential code with callbacks, and (4) the PTB-FLA code, respectively.

The input to phase 1 is the row sequential code and the output is the referent sequential code. The input row sequential code may come from various sources and may have various forms. Nowadays, many AI/ML algorithm solutions in Python are available online in Colab notebooks, where snippets of textual mathematical explanations, code snippets, and graph plots dynamically created by code play (i.e., execution) are interleaved. To make the output referent sequential code, a PTB-FLA developer essentially must select only the necessary code snippets, to tweak them if needed, and to integrate them into a standalone Python module(s) that they could preferably run on their PC (localhost), typically in a terminal of some IDE.

The important requirement for the referent sequential code is that for a given input dataset it must deterministically produce some output data e.g., learned (trained) model coefficients and/or some quality indicators like accuracy, because this output data is used as the referent data by the next development phases, which they must produce (with some small error) to be semantically equivalent. To that end, a PTB-FLA developer should use asserts that automatically compare whether the output data is (approximately) equal to the referent data, and if not, report the corresponding assertion error.

In phase 2, a PTB-FLA developer makes the federated sequential code by the following three steps: (1) partition the input dataset into partitions (that could be distributed across clients), (2) split the monolithic computing of the complete input dataset into a series of computing on individual partitions (that could be collocated with corresponding partitions) such that they produce the set of partition models, and (3) add the computing for aggregating the set of partition models into the final model and for computing quality indicators (that could be located on a server), as well as for comparing output and referent data. For example, a single function call (calling the function *f*) to process the complete dataset could be split into a series of function calls (calling the same function *f* with different arguments) to process individual dataset partitions. Note that this is still one sequential program that runs on a single machine (developer's PC).

In phase 3, a PTB-FLA developer makes the federated sequential code with callbacks by the following four steps: (1) copy (and tweak) the computing on an individual partition (say a partition number *i*) into the client callback function, (2) replace the series of computing on individual partitions with the series of client callback function calls (with the arguments corresponding to the partition *j* in the call number *j*), (3) copy (and tweak) the computing for aggregating the set of partition models to the final model into the server callback function, and (4) replace the former computing with the server callback function call (the code for computing quality indicators should remain in its place). In the running example, the series of function calls (calling the same function *f* with different arguments) to process individual dataset partitions should be replaced with the corresponding series of client callback function calls, which lead to indirect calls to the function *f* (each call to the client callback function maps to the corresponding call of the function *f*).

In phase 4, a PTB developer makes the PTB-FLA code by the following two steps: (1) add the code for creating the instance *ptb* of the class *PtbFla* and for preparing local and private data for all the instances, and (2) replace the code for calling callback functions (both the series of client callback function calls and the server callback function call) with the call to the function fl_centralized (in case of a centralized FLA) or fl_decentralized (in case of a decentralized FLA) on the object *ptb*.

Generally, a PTB-FLA developer should first develop the centralized FLA and then develop the decentralized FLA, because the centralized FLA is simpler and easier to comprehend.

4.1.2 Example of Implementation: Logistic Regression

The PTB-FLA development paradigm was validated and illustrated by the case study on logistic regression, wherein both centralized and decentralized FLAs were developed. The input code for phase 1 was the Colab notebook by Adarsh Menon [45], which uses the Social Network Ads (SNA) dataset. The next four tables show the outputs of the four phases (in Pythonic pseudocode) for the centralized logistic regression FLA (CLR-FLA). The output code for phase 1 (i.e., the referent sequential code) comprises the main function seq_base_case and two supplementary functions: logistic regression and evaluate, see Table 1.

Table 1: CLR-FLA phase 1 output code

```
// pd is representing the Pandas library
01: seq_base_case()
02:         data = pd.read_csv("Social_Network_Ads.csv")
03:         X_train, X_test, y_train, y_test = train_test_split(data['Age'],
                        data['Purchased'], test_size=0.20, random_state=42)
04:         b0, b1 = logistic_regression(X_train, y_train)
05:         y_pred, accuracy = evaluate(X_test, y_test, b0, b1)
// The supplementary function logistic_regression
06: logistic_regression(X, Y, b0=0., b1=0., L=0.001, epochs=300)
07:         X = normalize(X)
08:         for epoch in range(epochs)
09:                 y_pred = predict(X, b0, b1)
10:                 D_b0 = -2 * sum((Y - y_pred) * y_pred * (1 - y_pred))
11:                 D_b1 = -2 * sum(X * (Y - y_pred) * y_pred * (1 - y_pred))
12:                 b0 = b0 - L * D_b0
13:                 b1 = b1 - L * D_b1
14:         return b0, b1
// The supplementary function evaluate
15: evaluate(X_test, y_test, b0, b1)
16:         X_test_norm = normalize(X_test)
17:         y_pred = predict(X_test_norm, b0, b1)
18:         y_pred = [1 if p >= 0.5 else 0 for p in y_pred]
19:         accuracy = 0.
20:         for i in range(len(y_pred)):
21:                 if y_pred[i] == y_test.iloc[i]:
22:                 accuracy += 1.
23:         accuracy = accuracy / len(y_pred)
24:         return y_pred, accuracy
```

The output code for phase 2 (i.e., the federated sequential code) comprises the main function seq_horizontal_federated (see Table 2) and the supplementary functions from phase 1, and it targets the system with three instances (two clients and one server).

Table 2: CLR-FLA phase 2 output code

```
01: seq_horizontal_federated()
02:         data = pd.read_csv("Social_Network_Ads.csv")
03:         X_train, X_test, y_train, y_test = train_test_split(data['Age'],
                    data['Purchased'], test_size=0.20, random_state=42)
04:         X_train_0 = X_train.iloc[:160]
05:         X_train_1 = X_train.iloc[160:]
06:         y_train_0 = y_train[:160]
07:         y_train_1 = y_train[160:]
08:         b00, b01 = logistic_regression(X_train_0, y_train_0)
09:         b10, b11 = logistic_regression(X_train_1, y_train_1)
10:         b0 = (b00 + b10)/2.
11:         b1 = (b01 + b11)/2.
12:         y_pred, accuracy = evaluate(X_test, y_test, b0, b1)
13:         return [b0, b1]
```

Table 3: CLR-FLA phase 3 output code

```
01: seq_horizontal_federated_with_callbacks()
02:         data = pd.read_csv("Social_Network_Ads.csv")
03:         X_train, X_test, y_train, y_test = train_test_split(data['Age'],
                    data['Purchased'], test_size=0.20, random_state=42)
04:         X_train_0 = X_train.iloc[:160]
05:         X_train_1 = X_train.iloc[160:]
06:         y_train_0 = y_train[:160]
07:         y_train_1 = y_train[160:]
08:         localData = [0., 0.]
09:         msgsrv = [0., 0.]
10:         msg0 = fl_cent_client_processing(localData, [X_train_0, y_train_0], msgsrv)
11:         msg1 = fl_cent_client_processing(localData, [X_train_1, y_train_1], msgsrv)
12:         msgs = [msg0, msg1]
13:         avg_model = fl_cent_server_processing(None, msgs)
14:         b0 = avg_model[0]
15:         b1 = avg_model[1]
16:         y_pred, accuracy = evaluate(X_test, y_test, b0, b1)
17:         refbs = seq_horizontal_federated()
18:         assert refbs[0] == b0 and refbs[1] == b1
19: fl_cent_client_processing(localData, privateData, msg)
20:         X_train = privateData[0]
21:         y_train = privateData[1]
22:         b0 = msg[0]
23:         b1 = msg[1]
24:         b0, b1 = logistic_regression(X_train, y_train, b0, b1)
25:         return [b0, b1]
26: fl_cent_server_processing(privateData, msgs)
27:         b0 = 0.; b1 = 0.
28:         for lst in msgs:
29:                 b0 = b0 + lst[0]
30:                 b1 = b1 + lst[1]
31:         b0 = b0 / len(msgs)
32:         b1 = b1 / len(msgs)
33:         return [b0, b1]
```

The output code for phase 3 (i.e., the federated sequential code with callbacks) comprises the main function seq_horizontal_federated_with_callbacks (see Table 3) and the supplementary functions from phase 1.

The output code for phase 4 (i.e., the PTB-FLA code) comprises the main function ptb_fla_code_centralized (see Table 4), the supplementary functions from phase 1, and the main and the callback functions from phase 3.

Table 4: CLR-FLA phase 4 output code

```
01: ptb_fla_code_centralized(noNodes, nodeId, flSrvId)
02:         data = pd.read_csv("Social_Network_Ads.csv")
03:         X_train, X_test, y_train, y_test = train_test_split(data['Age'],
                    data['Purchased'], test_size=0.20, random_state=42)
04:         X_train_0 = X_train.iloc[:160]
05:         X_train_1 = X_train.iloc[160:]
06:         y_train_0 = y_train[:160]
07:         y_train_1 = y_train[160:]
08:         ptb = PtbFla(noNodes, nodeId, flSrvId)
09:         lData = [0., 0.]
10:         if nodeId == 0
11:                     pData = [X_train_0, y_train_0]
12:         else if nodeId == 1
13:                     pData = [X_train_1, y_train_1]
14:         else
15:                     pData = None
16:         ret = ptb.fl_centralized(fl_cent_server_processing,
                    fl_cent_client_processing, lData, pData, 1)
17:         b0 = ret[0]; b1 = ret[1]
18:         y_pred, accuracy = evaluate(X_test, y_test, b0, b1)
19:         if nodeId == flSrvId:
20:                     refbs = seq_horizontal_federated()
21:                     assert refbs[0] == b0 and refbs[1] == b1
22:         del ptb
```

Once CLR-FLA was developed, developing the decentralized logistic regression FLA (DLR-FLA) was rather straightforward, see the previously mentioned paper for more details.

4.1.3 Applications within TaRDIS use cases

PTB-FLA might be used by LSTM (Long Short-Term Memory) and DRFL (Deep Reinforcement Federated Learning) models to be jointly developed by NKUA and UNS, as well as by the ODTS (Orbit Determination and Time Synchronization) algorithm in the distributed ODTS simulation to be jointly developed by GMV and UNS. The former might be used in the EDP use case whereas the latter might be used in the GMV use case.

4.2 Personalized and clustered FL and implementations in the Flower framework

In this section, we describe the work on personalized and clustered federated learning, as well as their FL implementations, carried out in the current reporting period. Specifically, for personalized FL, we carried out the Flower-based [46] implementation and evaluation of a generally applicable state of the art method dubbed pFedMe [47]. The method is very general and may be applied to various ML tasks, including, e.g., supervised classification tasks of interest to TID use cases as described in Section 3.2. At an innovation level, we provided a new version of the method called pFedMeNew, described below. Regarding clustered FL, we provided methodological contributions by proposing a new method for one shot clustered learning. For details on the method and its evaluations, we refer to [48].

4.2.1 Overview of Flower framework

Flower [46] represents an open-source FL framework that supports executing large-scale experiments efficiently. Also, it is adaptable to heterogeneous FL device scenarios. Flower is flexible regarding different languages and machine learning frameworks and highly extensible. Its main goal is to unify FL research and real-world FL systems, while offering tools for defining ML models and the basic skeleton for building FL strategies. Flower provides a stable, language and ML-framework agnostic implementation of the core components of a FL system.

The architecture of the framework implies the existence of Server and Client abstractions, while in between lies the Strategy abstraction that directs the FL process. Flower uses bidirectional Google Remote Procedure Calls (gRPC) streams as a communication protocol by default. This approach enables message exchanges serialized using efficient binary formats, which is very useful for low-bandwidth devices. The FL process begins by starting the server. At the server's side, 3 major components can be identified: the ClientManager, the FL loop and a Strategy. The default strategy is FedAvg, but it can be customized as needed. The clients are sampled from the ClientManager. It manages a set of clients, ClientProxy objects. The FL loop orchestrates the FL process, by collaborating with the Strategy. The clients are waiting for messages from the server. A client first receives the model parameters from the server, then it starts the local model training. When the training is finished, the client generates the updated model parameters and sends them back to the server. Beside real edge clients, Flower also supports Virtual Clients, which is very useful for running quick simulations without having available resources on a single machine. Figure 14 illustrates the described Flower architecture.

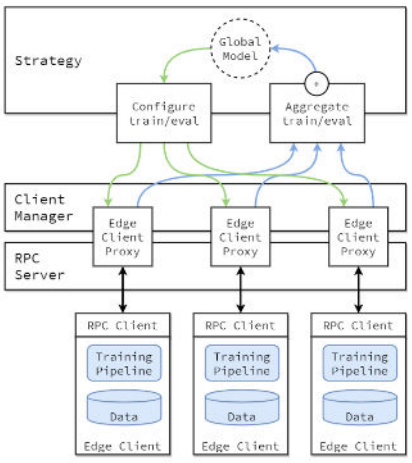

Figure 14: Flower core framework architecture [48]

Flower satisfies some important design goals, including: scalability (scales to a possibly large number of clients), client-agnosticism (interoperable with heterogeneous clients), communication-agnosticism (allows different serialization and communication approaches), privacy-agnosticism (supports common privacy requirements), flexibility and expandability (easy change and adoption). These goals are driven by the need to handle diverse client environments, keep track with the rapid advancements in FL, simplify the complex implementation process of an FL system, and seamlessly integrate with evolving ML frameworks.

4.2.2 Flower FL implementations in TaRDIS

The algorithms within Task 5.1, in the FL training models API will use the structure that the Flower framework offers, while implementing customized strategies for ML algorithms. The implementations for FedAvg and Personalized Federated Learning with Moreau Envelopes have been developed so far. Supervised FL can be viewed as an empirical risk minimization problem, where a number of devices is participating in training the global model. A popular and commonly used solution is the FedAvg [49] algorithm. The main disadvantage of FedAvg is poor performance guarantee compared to non-local counterparts when there is data heterogeneity [50]. Data heterogeneity represents a common problem regarding the usability of a global model to individual users. There are different mitigation strategies for this problem. One of them is personalized FL [51] that is of interest here. This approach observes FL as a mixture of global and local models (referred to as personalized models).

Personalized FL with Moreau envelopes [47] (pFedMe) uses Moreau Envelopes as client's regularized loss functions. The approach ensures that the personalized model on the client remains close to the global model. The significance of the local model concept becomes prominent. This is achieved by utilizing a regularized loss function that incorporates an L2 norm-based penalty for each client:

$$\min_{x_1,\ldots,x_n \in \mathbb{R}^d} f(x) + \lambda\psi(x) := F(x)$$

$$f(x) := \frac{1}{n}\sum_{i=1}^{n} f_i(x_i) \quad \text{and} \quad \psi(x) := \frac{1}{2n}\sum_{i=1}^{n} \|x_i - \bar{x}\|^2$$

Here, $x_1,\ldots,x_n \in R^d$ represent the local models, while the mean of local models is defined as:

$$\bar{x} := \frac{1}{n}\sum_{i=1}^{n} x_i$$

The parameter λ is a penalty parameter, which controls the difference from the mean. With λ=0, we are talking about local models, while when λ=∞, then we have a global model. The value $\lambda \in (0, \infty)$, refers to mixed models.

The pFedMe algorithm is implemented by using the Flower framework and PyTorch. Additionally, a slightly modified version of pFedMe, the pFedMe_new algorithm has been also implemented. These two implementations differ in local iterations and batch sampling. Both algorithms perform a defined number of global rounds, and a defined number of local rounds. Inside the local rounds, the algorithms run K inner iterations. The novel algorithm pFedMe_new takes a new batch inside every inner iteration, while pFedMe, takes a batch and performs the inner iterations on it. The pseudocode for both algorithms can be written as follows:

Funded by the European Union

pFedMe_new

k = 0, . . . , K−1

Sample a batch D_i

Perform a step w.r.t. D_i

$$\tilde{f}_i(\theta_i; \mathcal{D}_i) + \frac{\lambda}{2}\left\|\theta_i - x_{i,r}^t\right\|^2$$

After K iterations obtain $\tilde{\theta}_i(x_{i,r}^t)$

Compute the new local model: $x_{i,r+1}^t = x_{i,r}^t - \eta\lambda(x_{i,r}^t - \tilde{\theta}_i(x_{i,r}^t))$

pFedMe

Sample a batch D_i

k = 0, . . . , K−1

Perform a step w.r.t. D_i

$$\tilde{f}_i(\theta_i; \mathcal{D}_i) + \frac{\lambda}{2}\left\|\theta_i - x_{i,r}^t\right\|^2$$

The numerical evaluations were performed in a homogeneous and heterogeneous setting. One of the main aims is to validate the effectiveness of implementations of pFedMe and pFedMe_new algorithms in a heterogeneous setting, but also the validation of the advantages of FedAvg in a homogeneous setting. The experiments also highlighted the limitations of independent learning in comparison to FL, as meaningful learning cannot be attained independently by a client with a relatively small, but diverse dataset and a compact model. The evaluation was performed on the FashionMNIST data set, where the task is image classification with 10 classes. The experiments took place on a University of Novi Sad's computer cluster infrastructure. The model utilized a compact CNN (0.331 MB) with added batch normalization layers. For all the tests, 10 clients and a central server were used. For the homogeneous setting, the complete FashionMNIST dataset is evenly partitioned among 10 clients, where each client has 5400 training samples and 600 testing samples. The heterogeneous setting is as follows: each client has a unique set of classes, where the training set size is between 4000 and 6000 samples and the test set size is 10% of the training set; the clients have non-overlapping class sets (Client 1 has [0,1,2,3,4], Client 2 has [1,2,3,4,5] and so on). Regarding the experimental settings, in the context of pFedMe and pFedMe_new, the local rounds refer to the number of times K inner iterations are performed (this should not be confused with local epochs, which typically refer to going through the entire training set, like in fedAvg). We use 100 global rounds, 120 local rounds and 10 inner iterations for pFedMe and pFedMe_new evaluations. For FedAvg, we use 100 global rounds and 1 local epoch.

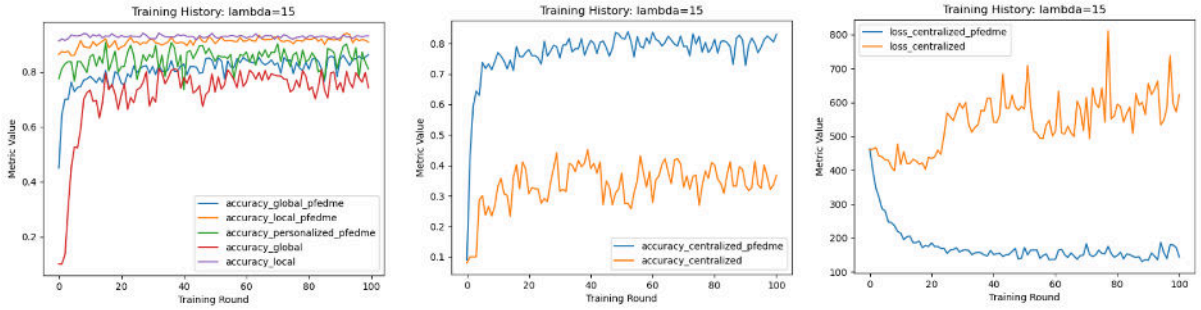

Figure 15: Accuracies of clients (left), the server (middle) and loss of the server (right) in the heterogeneous case

Figure 15 (left) shows the average accuracies of the 10 clients. In pFedMe, each client first evaluates the accuracy of the personalized model (accuracy_personalzed_pfedme on Figure 15, left). Then, the clients evaluate their local models on their own test data (accuracy_local_pfedme for pFedMe and accuracy_local for FedAvg on Figure 15, left). The server evaluates the global model on the whole test data set (accuracy_global_pfedme for pFedMe and accuracy_global for FedAvg on Figure 15, left). According to this, the best model for the client is the local model obtained by FedAvg (accuracy local), but it can be seen that the global model from pFedMe (accuracy global pFedMe) shows better

performance than the global model from FedAvg (accuracy global). Figure 15 (center) shows that the global model from pFedMe produces higher accuracy than the global model from FedAvg. The decreasing loss on Figure 15 (right) proves the stability of pFedMe.

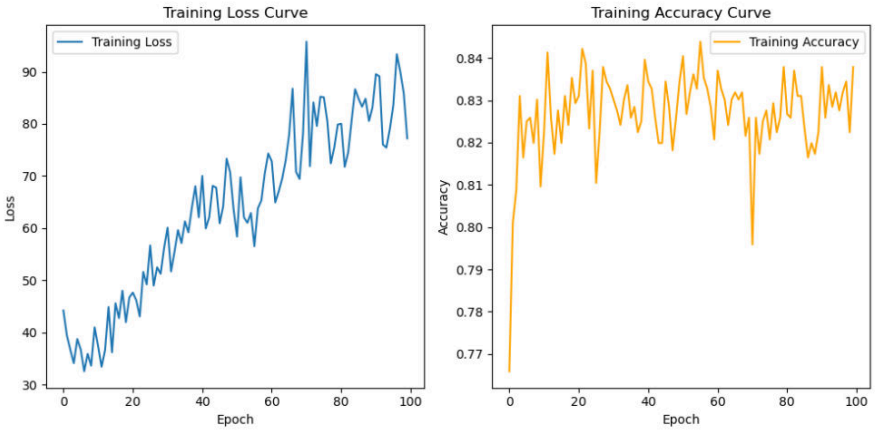

Figure 16: A client trained in isolation in the heterogeneous case

The instability in loss and accuracy when training a model in isolation is shown on Figure 16. It proves that the model cannot be improved while being trained alone, but instead collaborative training leads to an overall improvement.

In a homogeneous setting, the FedAvg algorithm outperforms the pFedMe algorithm. However, pFedMe excels in generating a robust global model that is suitable for heterogeneous settings.

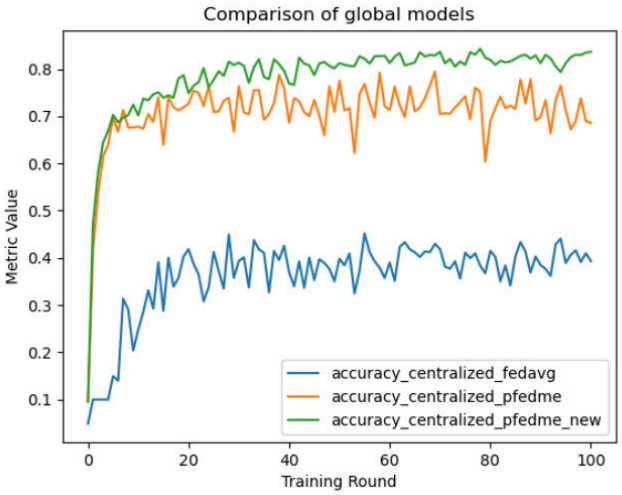

Figure 17: Global model accuracies from FedAvg, pFedMe, and the new implementation of pFedMe

Finally, Figure 17 shows the accuracies for the global model for the discussed implementations. We observe that the pFedMe_new algorithm performs the best.

The list of algorithms is currently evolving and will expand according to the needs of the use cases. It will offer a solution for anomaly detection with noisy labels, that is of particular interest for the ACT use case. Also, distributionally robust FL algorithms will be also considered.

Funded by the European Union

4.3 Discussion

Task 5.1 needs to provide a framework supporting AI/ML programming primitives. The initial steps towards this goal have already been taken, as described in subsections 4.1 and 4.2. Also, other algorithms that might be of interest are already identified, and their development is in progress. These actions represent the base for the development of a decentralized AI/ML. Further specification and advancements of computing primitives and other aspects of the framework towards generalization will be made according to the appearing needs. The AI/ML library will therefore gradually evolve, and the readily deployable code examples will be added accordingly. This will include some common ML and inference tasks (personalized federated learning for example), as well as use case specific solutions (anomaly detection of workflows with noisy labels for ACT). The classification of code to the different execution environments: swarm and device-edge-cloud, will emerge naturally with the algorithms implemented.

The development of an optimisation-based module for configuring primitives to the constraints will be covered in a later phase of development. The development of a platform for federated learning algorithms is an ongoing task that currently includes the work on PTB-FLA and Flower-based solutions, described in subsections 4.1 and 4.2, whose completion is envisioned for the later phases of the project, when a mature set of developed algorithms is available and tested.

Funded by the European Union

5 Advances on AI-driven planning, deployment and orchestration framework

5.1 Simulator for computer networks

Reinforcement Learning (RL) algorithms require an interactive environment to learn. As our focus is on task offloading in a peer-to-peer (P2P) setting, we need to create a P2P network to train various RL agents. While deploying the system in a real-world P2P network would best capture its unpredictability and complexity, these networks often handle enormous amounts of participants. This leads to high costs and complexity in development and deployment. Therefore, using simulations is a practical way to provide an environment for RL without the time and expense associated with deploying real networks.

To explore various paradigms for task offloading with distinct features, we need a flexible and conductive simulation. It should support different offloading paradigms and scale to a large number of devices while accommodating network dynamics, including node mobility and churn. For this, we chose to use the PeerSim simulation tool, a Java-based tool designed for simulating large-scale dynamic P2P networks [52].

In addition to the simulation, we defined an environment using the PettingZoo Python framework [53]. This allows us to train agents by wrapping the simulation as a Markov Decision Process. The environment manages the interactions between the agents and the simulation, enabling the effects of these interactions to take place within the simulation.

5.1.1 System Model and Assumptions

5.1.1.1 Task Model

All output from the simulation comes in the form of applications, each consisting of a group of tasks. Our goal is to enable various types of simulations, each generating applications with unique properties based on the specific simulation implemented. For Binary-Online and batch offloading simulations, an application comprises a single task. However, in a simulation with dependencies, an application is made up of a Directed Acyclic Graph (DAG) of tasks.

A task consists of a tuple $\tau_c = \langle c, I, T_{in}, T_{out}, CPI, D \rangle$, where:

- c - Identifier of the task
- I - Amount of instructions to be processed,
- T_{in} - Total data size inputted,
- T_{out} - Total data size produced,
- CPI - Cost in CPU cycles per instruction.
- D - Deadline of the task (Can be set to infinity, if there is no deadline)

An application consists of a DAG of tasks, where each task represents a single node of the DAG. The model of an application is $A = (T, D)$:

- $T = \{t_1, \ldots, t_m\}$ is the set of tasks acting as the vertices of the graph
- D is the set of dependencies between vertices in the graph, such that a dependency is represented as an ordered pair, $d = (t_n, t_m)$, where t_n is a predecessor task of t_m.

5.1.1.2 System Model

Our simulation is built to handle various configurations, accommodating both peer-to-peer simulations and those with an imposed hierarchy of nodes, with or without cloud access. We will also consider node mobility in the next phase.

We construct these configurations with two types of nodes:

- **Worker Nodes:** These nodes simulate task execution as requested by clients. They have a processing limit; when exceeded, they become overloaded and begin to drop new tasks. Each worker node may run a controller protocol, enabling the node to offload tasks to other workers. Any worker that completes a task sends an answer back to the original worker responsible for the task's application.
  Each node can execute a variable number of instructions per time step. Once enough instructions to complete a task have been executed, the task is deemed complete. When all tasks in an application are complete, the worker sends the results back to the client that initially requested the application. Workers periodically broadcast any changes to their internal state to inform neighboring workers, enabling accurate offloading decisions. The controller protocol manages these broadcasts.
- **Client Nodes**: These nodes generate tasks and send them directly to the worker nodes for execution. Task generation follows a Poisson process. Each client maintains an independent Poisson process for each worker node, all with the same mean event rate. Each client also tracks the average request time for analytical purposes. In our preliminary work, we consider only one task type, meaning every task will have the same data size and number of cycles.

The network topology is fully configurable, supporting both random network generation and manual placement of nodes. Links between nodes can also be manually set or connect all nodes within a user-defined radius. To allow node heterogeneity, we enable the definition of multiple node categories and a variation in processing power within each category. This mechanism also allows the definition of node hierarchies, with each category of nodes representing a distinct network layer, and nodes more than one layer apart remaining unlinked.

A diagram of a hierarchical network can be seen in Figure 18.

The simulation is divided into discrete time steps. To facilitate agent-simulation interaction, we run a protocol that manages action distribution across multiple controllers within the simulation, which then execute these actions and manage simulation pause and continuation.

Funded by the European Union

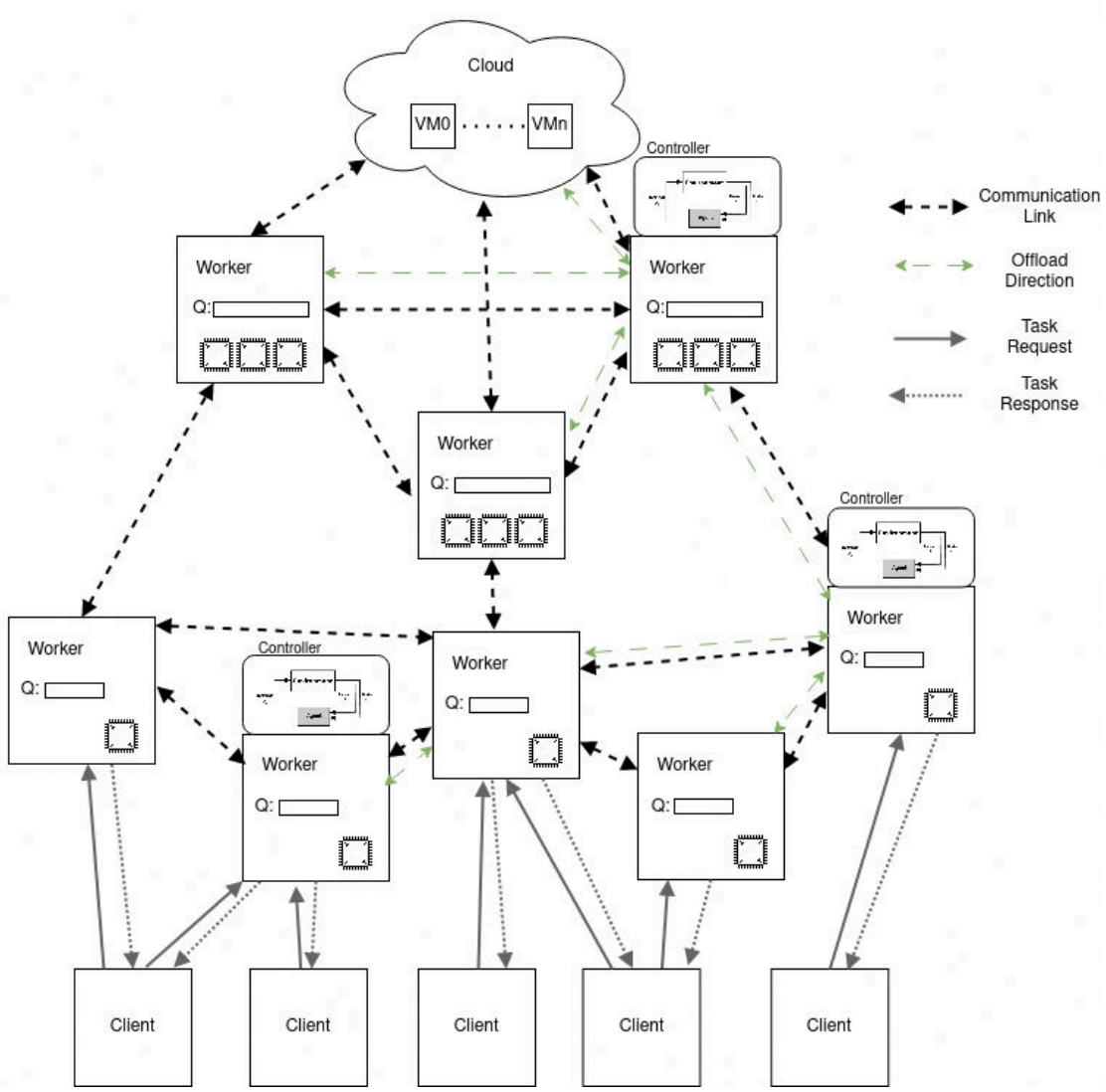

Figure 18: Model of a hierarchical network

5.1.1.3 Wireless Communication Model

Variables for the communication model follow the notation of Baek et al [54]. We utilize the Shannon-Hartley Theorem to compute the delay in sending tasks wirelessly to another node, having the delay in seconds, t^c, of transmitting T bytes from node i to node j as:

$$t^c = T/r_{l,o}$$

$$r_{i,j} = B\ log\left(1 + \frac{g_{i,j}P_{t_x,i}}{BN_0}\right)$$

Where:

- B – Bandwidth of the outgoing link (Mhz)
- $g_{i,j}$ - Channel gain
- N_0 - Noise spectral density (kept a constant 174 dBm/Hz)
- $P_{t_x,i}$ – Transmission power of node i (dBm)

- $r_{i,j}$ – is the channel capacity in bits per second, i.e., a tight upper-bound on the maximum rate of information that can be transferred with a minimal error rate, using an average received signal power $g_{i,j}P_{t_x}i$, over an analog communication channel affected by additive white Gaussian noise (AWGN) with power BN_0.

5.1.2 Environment

The simulation alone is insufficient for training a Reinforcement Learning (RL) agent. We also need to create an environment following the Markov Decision Process (MDP) framework. For this, we use the PettingZoo framework, that standardizes environment creation.

5.1.2.1 The Markov Decision Process (MDP)

In the initial phase, a single centralized controller distributes the current workload across a network of machines. This controller uses RL to decide whether to offload tasks arriving at the worker or process them locally. The RL agent makes this decision by examining the state of the worker and the information it has about its neighborhood.

This section introduces the MDP for the centralized case. In the next version, we will adapt the model to allow for multiple agents in a federated fashion. At that point, we will upgrade the formulation to a Decentralized-MDP or a Markov Game. A Markov or stochastic game, introduced by Lloyd Shapley in the 1950s [55], is a repeated, probabilistic game played by one or more players across sequential stages. Each stage starts in a state where players select actions and earn payoffs based on these actions and the state. The game then transitions to a new random state, influenced by the previous state and actions. This process continues for either a finite or infinite number of stages. The total payoff is often the discounted sum of stage payoffs or the limit inferior of the stage payoffs' averages. Stochastic games extend Markov decision processes to multiple decision-makers and strategic-form games to dynamic environments that respond to player choices.

An MDP is represented as a tuple of the form $\langle S, A, P, R, \gamma \rangle$, following the convention of Baek et al. We now define each element of our MDP, for a network with N nodes:

- State Space, S, is the set of all possible states

  $S = \left(N, L, Q, Pwr, P, B_{max}, T_{pwr}\right)$.

  - N - An array with the identifiers of all the nodes in the network; all the other arrays respect the order of the IDs in this array.
  - L - An array storing the layer/tier of every node (lower, middle, or upper)
  - Q - An array storing the last known queue size for each node in the network.
  - Pwr - An array storing the latest known processing power for each node in the network.
  - P - An array storing the known position for each node in the network.
  - B_{max} - An array storing the bandwidth of the channels of the centralized node to all its neighbors.
  - T_{pwr} - The transmission power of the central node's antenna
- Action Space, A, is the set of all possible actions for the centralized node, deciding to offload, represented as $a = \left(n^o\right)$.

  - Variable n^o, $n^o \in N$, $1 \leq n^o \leq N$, is the index of the offload target node in the node's neighbor list, a task is to be processed locally whenever $n^o = 0$.
- Transition Function, P, is the unknown transition function.
- Reward Function, R, is structured to maximize a utility function, $U(s,a)$, and minimize the total delay, $D(s,a)$, and the overload cost, $O(s,a)$. The reward for the action a in state s is given by:

$$R(s,a) = U(s,a) - (D(s,a) + O(s,a))$$

  - The Utility function, $U(s,a) = r_u log\left(1 + w^l + w^o\right)$ represents the gain over completed tasks (by taking the action a in state s), where the r_u is the utility reward.
  - The Delay function, $D(s,a) = \chi_d\left(t^w + t^c + t^e\right)$ has three components:
    - χ_d is the delay weight
    - $t^w = \frac{Q^l}{\mu^l} w^l + \left(\frac{Q^l}{\mu^l} + \frac{Q^o}{\mu^o}\right) w^o$ is the average time a task will be sitting in the queue of the fog node n^l, where μ^i is the computing service rate of node n^i. This is similar to the service rates on queue theory, being measured in tasks processed per unit of time. w^o is the number of tasks to be offloaded (either 1 or 0). w^l is the number of tasks to be processed locally (either 1 or 0).
    - $t^c = \frac{T * w^o}{r_{l,o}}$ is the communication cost in time of offloading w^o tasks, where T is the data size of the tasks to be offloaded from n^l to n^o. The cost of the delay in sending a task is explained in section 5.1.1.3
    - $t^e = \frac{I \cdot CPI \cdot w^l}{f^l} + \frac{I \cdot CPI \cdot w^o}{f^o}$ is the last element of the delay function and represents the execution cost of the tasks on the offloaded nodes. Here, I is the average number of instructions per task, CPI is the average number of CPU cycles per instruction of the nodes, and f^l and f^o are the total instructions per second the workers are capable of processing.
  - The last element being considered in the reward function is the cost of overloading any of the nodes involved in the offloading, $O(s,a) = -\chi_o \, log\left(w^l P_{overload,l} + w^o P_{overload,o}\right)$, where:
    - χ_o is a constant factor representing the weight of overloading
    - $P_{overload,i} = max\left(0.00001, \frac{\left(Q^i_{max} - Q'_i\right)}{Q^i_{max}}\right)$, represents the distance to overloading n^i
      - $Q'_i = min\left(max\left(0, Q_i - \mu^i\right) + w^i, \; Q_{i,max}\right)$ is the expected state of the queue at node n^i after taking action a.
- Discount Factor, γ, is the discount factor that will be used to discount the expected future rewards.

5.1.2.2 The Environment

As previously mentioned, we construct our environment in Python using the PettingZoo framework, which interacts with the simulation. To bridge the gap between the Python environment and the Peersim Simulation running in the Java Virtual Machine (JVM), we use Spring Boot Server. This platform exposes a REST API for posting actions and retrieving information about the simulation. The full stack of the environment is demonstrated in Figure 19.

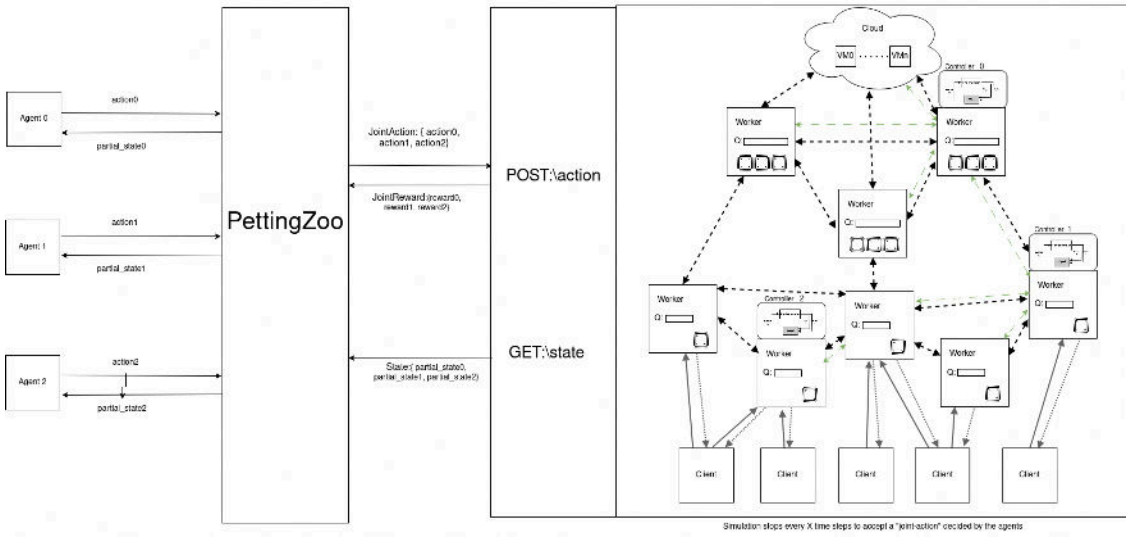

Figure 19: The full stack of the environment

The agents' interaction with the environment begins when the simulation pauses to wait for an action. The agents observe the latest state available in the environment, decide on an action based on these observations, and send it to the environment using the available step function. The environment collects the actions taken by all the agents and processes them into a POST request, which it sends to the simulation. After the actions are sent and the simulation resumes, the environment waits for the simulation to pause again, fetches the latest state, and the cycle repeats.

One of PettingZoo's most valuable features is its ability to use hooks to visualize the environment[1]. To aid our agents' development, we implemented a simple visual rendering using Pygame. Figure 20 displays the rendering of a basic network, where each node is depicted as a rectangle with its ID in the green square at the upper-left corner. Within the rectangle, you can find details about its queue and the layer/category to which the node belongs. Notably, when a node offloads a task, the link used for the offloading flashes red, allowing us to quickly understand the agent's current behavior.

[1] In the context of software development, "hooks" are a type of function that allow developers to insert or "hook" their own custom code into existing functions or processes.

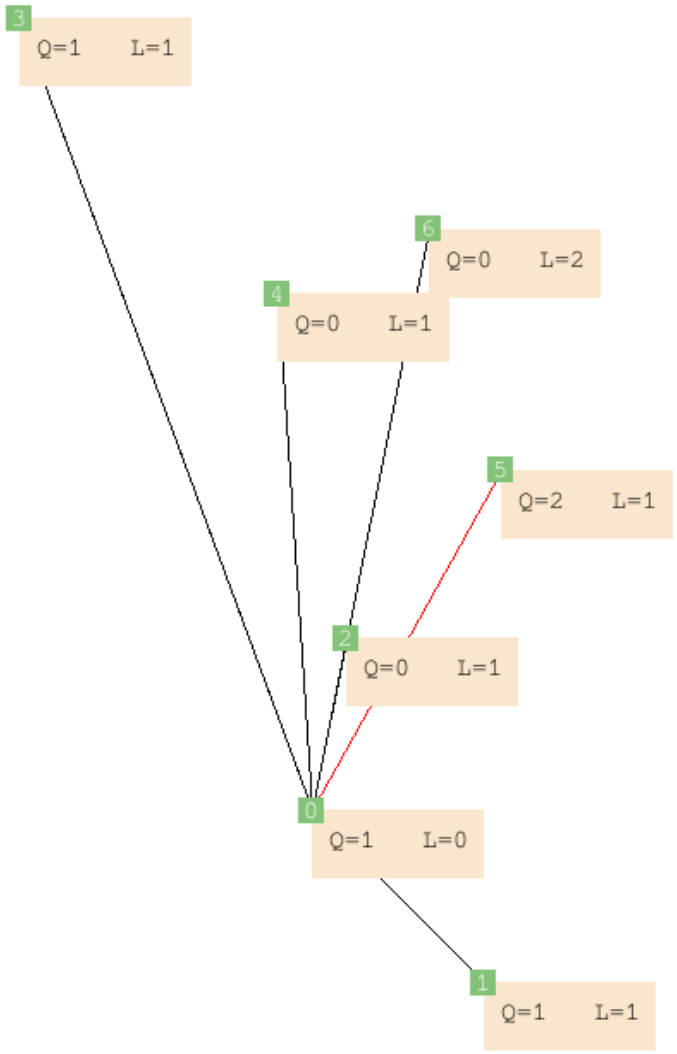

Figure 20: The rendering of a basic network

5.2 Centralized RL agents using the simulator

5.2.1 The Basics of Deep Q-Learning

Reinforcement learning is a method where an algorithm learns the best course of action in a given situation to maximize a reward. This learning is achieved through trial and error, without explicit direction, and considers not only immediate outcomes but also future rewards. For instance, a chess-playing algorithm using reinforcement learning would deduce the optimal move not just for immediate advantage, but considering future board positions and potential victories.

The typical process of a Reinforcement Learning (RL) model is illustrated in Figure 21. An agent interacts with an environment over multiple time steps, represented as $t \in Z_0^+$, aiming to accomplish a goal. At each time step, the agent observes the current state S_t, takes an action A_t, receives a reward R_{t+1} corresponding to the action, and observes the state of the

environment after taking the action, S_{t+1}. Then the agent adjusts its behavior based on the received reward.

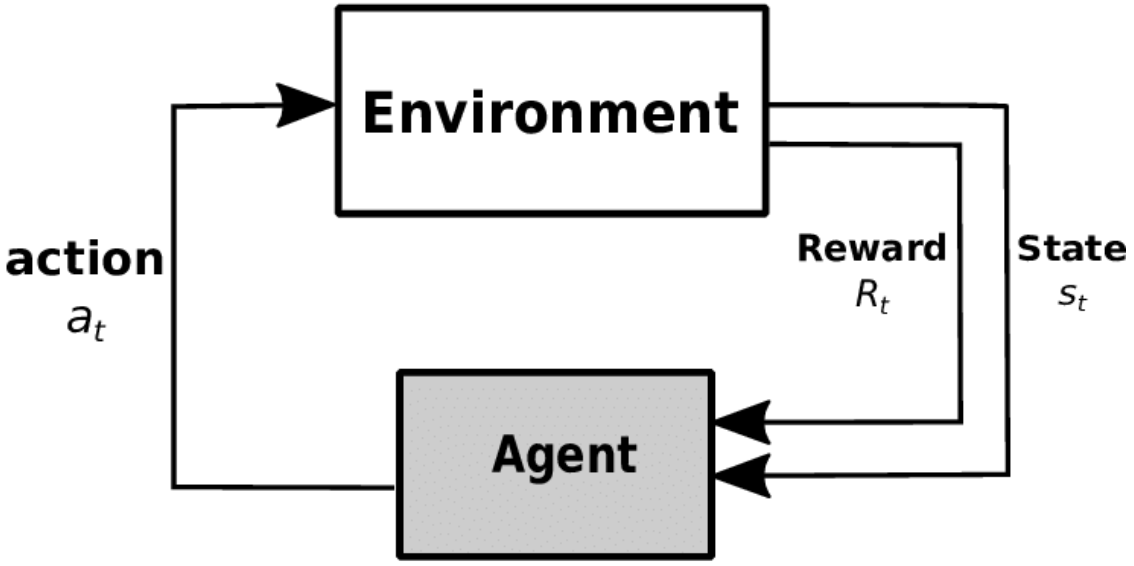

Figure 21: Reinforcement Learning Loop

A policy is a guideline that dictates a learning agent's behavior at a certain moment. Essentially, it's a rule that links perceived environmental states to corresponding actions. This rule can be linked to a set of stimulus-response associations in psychology. Policies can range from simple functions or lookup tables to complex computations like search processes. The policy is crucial in a reinforcement learning agent as it solely determines the agent's behavior. Typically, policies can be stochastic, defining probabilities for each action.

Training the agent requires a method to evaluate the quality of its actions, considering not just immediate outcomes, but future consequences as well. Additionally, we need a way to assess the effectiveness of different policies for comparison. This is achieved through the use of State-value Functions and/or Action-value Functions.

The state value function or V-function, $V_{\pi}(s)$, is a metric of how good a given state is given policy π, which usually consists of a prediction of the potential reward we can get from that state by following the policy π. The state value function is given by the expression:

$$V_{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s',r} P(s', r|s, a)\left[r + \gamma V_{\pi}(s')\right]$$

The Action-value Function or Q-function, $Q_{\pi}(a, s)$, similarly to the State-value function also measures potential reward, but not instead of measuring it for all actions on a state it considers how good taking action a form state s. The action value function is given by the expression:

$$Q_{\pi}(s, a) = \sum_{s',r} P(s', r|s, a)\left[r + \gamma V_{\pi}(s')\right]$$

We can then use the Q-value and V-value functions to do policy improvement. This is done by iteratively updating the value function using the expression:

$$V_{\pi'} = argmax_a \sum_{s',r} P(s,a)\left[r + \gamma V_{\pi'}(s')\right]$$

The ultimate goal of policy improvement is achieving an optimal policy, in the sense that it maximizes the total expected reward.

5.2.2 Double Deep Q-Network

A Deep Q-Network (DQN) is an algorithm that learns to approximate the Q-value function. In its essence the DQN works by predicting the Q-values for each possible action given a state, and by picking the action with the highest predicted Q-value we can expect the highest return. Usually, to keep learning new strategies that may bring better results than the estimated ones, we use an epsilon-greedy algorithm that with epsilon probability takes a random action. Generally, this epsilon starts with a high value that decreases as the training advances and our network learns more about the environment.

To train a Neural Network we require a loss function, the loss function will act as the "distance" towards a more correct result. In the case of DQN this loss is computed for the last state after receiving the reward and knowing what the next state is, and is usually the Mean-Squared Error between the predicted Q-value and a target which is computed with the expression:

$$Target = R + \gamma max_a Q(S', a)$$

where the R is the received reward and the S' is the next state.

The disadvantage of DQN, is that it is a bootstrapping algorithm, meaning that we are estimating the Q-values of one state based on an estimation of the Q-values for future states. This brings instability to the algorithm. Therefore we use a Double Deep Queue Network (DDQN) algorithm, that uses two mechanisms to help stabilize the training, memory replay, and a lagging Q-Network, which is the name of the networks approximating the Q-values.

The "Replay memory" or "experience replay" works by storing past experiences of the agent and randomly sampling from the experience to break the correlations between the data being used to train the agent, a direct upgrade to this technique is the prioritized replay where the most relevant experience is sampled more often. DDQN further stabilizes the training by employing two Q-networks, one to evaluate the Q-Values and another lagging Q-network used to compute the target values differently.

5.2.3 Current State of Development

We have implemented a DDQN which is currently in the process of being debugged. We have developed a set of mechanisms to help the development including a dataset generation module for warm-starting the agent. Furthermore, we have designed a suit of configurations of progressively higher complexity that have been benchmarked with our baseline policies to help in the process of developing and debugging the agents before they are set to learn simulations of real-life networks.

In the near future, we will explore the Duelling Q-Network where the network diverges into two branches, one to predict an Advantage action value, A(s, a), and another the State-value. These branches are merged in a special aggregation layer into the Q-value, generalizing learning across actions.

And we will possibly look into other types of algorithms for RL like policy-based algorithms. Furthermore, we wish to explore graph neural networks to deal with rapidly changing neighborhoods and highly dynamic networks.

To measure the effectiveness of the algorithm we built we intend to compare it to a set of baseline policies, namely:

- Always Local Processing - This policy never offloads tasks by always selecting to offload 0 tasks to the node with the lowest Q size.
- Randomly Offload - This policy will pick both the target of the offloading and the number of tasks to offload randomly.
- Least Queue - This policy naively selects the observable worker with the smallest queue size, then divides the origin node's offloadable tasks between the selected and origin worker.

5.3 Discussion

5.3.1 Timeline

For the period ahead, we hope to have a trained DDQN agent, that we will benchmark against all the baselines on static networks. Also we will start working on graph neural networks to deal with dynamic networks and changing neighborhoods.

The next step is to introduce Federated Learning (FL). We will start by developing a Client-Server approach to FL. The client-server approach to Federated Learning (FL) is a centralized model where a server coordinates the learning process. Multiple clients (which could be devices or nodes with data) independently compute updates to the model based on their local data and send these updates to the server. The server then aggregates these updates to improve the global model. This process is iterated until the model reaches a satisfactory level of accuracy. This approach allows for efficient use of distributed data sources, while maintaining data privacy as the raw data never leaves the client device. Once this solution is operating effectively, we will progress to a fully decentralized Peer-to-Peer (P2P) approach to FL.

6 Advances on lightweight, energy-efficient ML techniques

In this section, three techniques for making a ML model more lightweight and energy-efficient are presented. Following modern trends, all three techniques are related to the use of Deep Neural Networks (DNNs), since lightweight methods for trivial ML models (e.g. Support Vector Machines, Random Forest, etc.) are currently out of scope, since the DNNs require the most computational resources for operation at swarm systems or at swarm devices. The three methods are (i) early exit of inference techniques; (ii) knowledge distillation; (iii) pruning.

6.1 Early exit of inference techniques

As depicted in Figure 22, it is obvious that both pictures contain dogs to a human eye. To a computer, however, the discerning or categorization of these pictures might not be as obvious. Even if it can correctly classify them, the second one is far more apparent.

Figure 22: Hot dog and dog pictures from https://stock.adobe.com/gr_en/search/images?k=dachshund+hot+dog&asset_id=616733889 and https://www.britannica.com/animal/dachshund respectively

In particular, when these images are passed through all the layers of a DNN during the inference phase, they demand the same computational power, time, and energy. The early exit method takes advantage of the inhomogeneity in the difficulty of the classification task, to preserve computational resources and inference latency.

More specifically, additional exit layers are added in the middle of the network during its training phase. When a branch is reached in the neural network, the exit pathway is selected if the confidence of the network exceeds a specific threshold (Figure 23). If the instance is correctly classified in an earlier exit, we don't have to go through the entire network, thus saving lots of resources. It must be noted that, in the harder cases that reach the final exit (low confidence level in previous early exits), more time and energy is required than the original (without early exit branches) model. The model is trained by comparing the exits with the actual labels, as one would with any classification problem. As backpropagation happens from the layer where the loss is computed, the exits closer to the start help solve the vanishing gradient problem [56].

Funded by the European Union

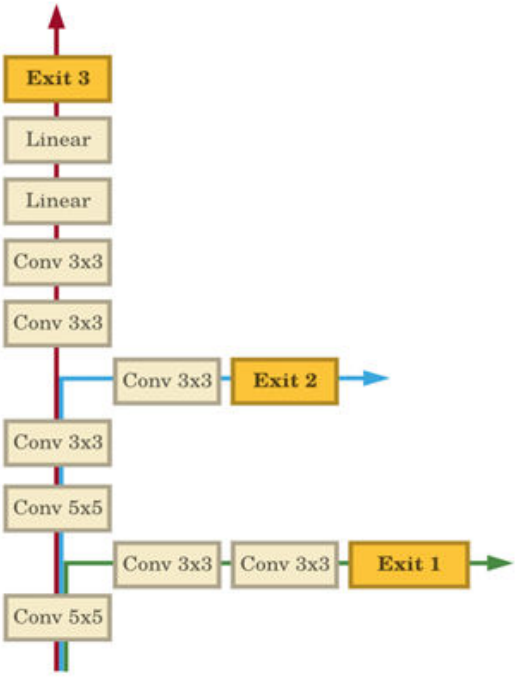

Figure 23: Early exit basic concept

6.1.1 Distributed Early Exit

The majority of edge devices are not able to infer the whole model locally due to resource constraints and latency requirements. Offloading to the cloud is a commonly used technique but comes with several disadvantages: (i) firstly, the amount of traffic that a network can support is limited; (ii) secondly, sending a message and waiting for a response has an inherent delay, regardless of how fast the model is. A solution that mitigates these downsides is partitioning the model between the local device and the cloud. Early Exit (EE) conceptually, partitions the model into sub models [57] [58] [59]. The model up to the first exit (or any exit we desire) can be stored and computed locally. Like normal early exit, we compare it to the threshold and if it does not exceed the selected threshold, the output results of the model's intermediate layers so far are conveyed to an edge server and subsequently to the cloud, as depicted in Figure 24. This enables the latency reduction due to message transmission, as well as save bandwidth on average. To this end, one can benefit from the regular advantages of EE, as well as possibly decrease the inference latency, in cases that we have less messages transmitted (compared to the case that we offload everything to the cloud).

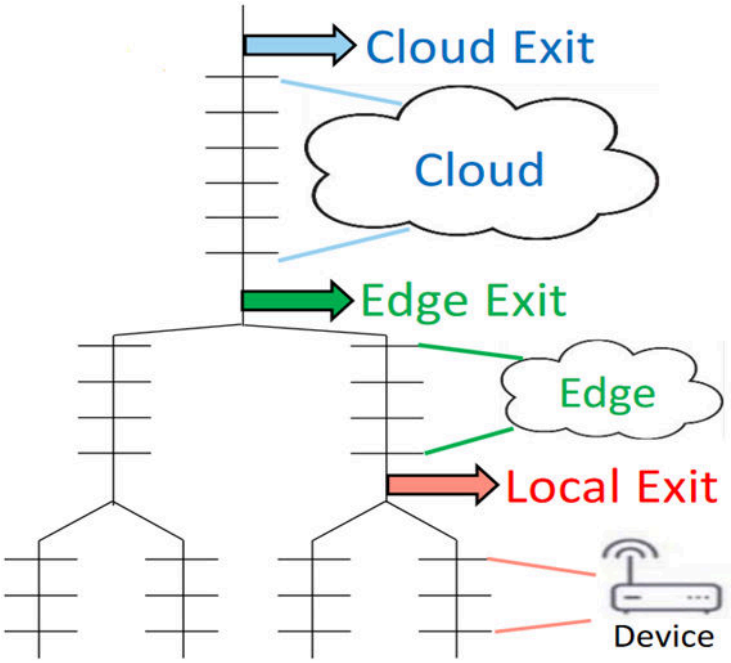

Figure 24: Distributed Early Exit according to Teerapittayanon [59]

6.1.2 Security in the Early Exit Framework

A very important advantage of the EE method is the security that it provides. The devices never send the data that is locally stored, but rather the output of an intermediate layer to the next neural network node. These outputs are, as most neural networks, nonsensical and, to a human seemingly random. They are practically encoded and, without having access to the local network itself, it is very hard to decrypt it, due to the random nature of neural networks.

6.1.3 Federated Learning and EE methods

The combination of EE and FL can be performed in an IoT device and cloud architecture, with each IoT device exhibiting computational constraints and containing a truck model (first shared layers between device and cloud model). The little branch is the first exit (Figure 25), whereas the big branch is the rest of the model's hidden layers.

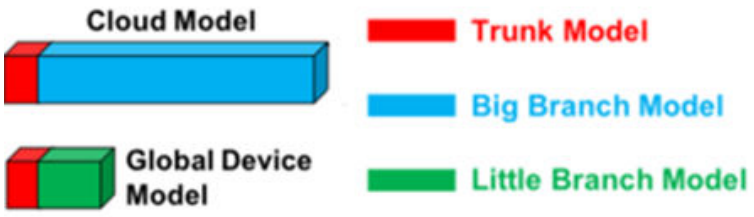

Figure 25: Early exit technique implemented in a Federated Learning framework [57]

The trunks and little branches of all devices are gathered, averaged, and distributed in each one. The big branch is trained and stays in the cloud. This idea can be extended to multiple nodes, with each one aggregating the models below it and to multiple architectural layers, for instance in an IoT-Edge-Cloud architecture, as depicted in Figure 26.

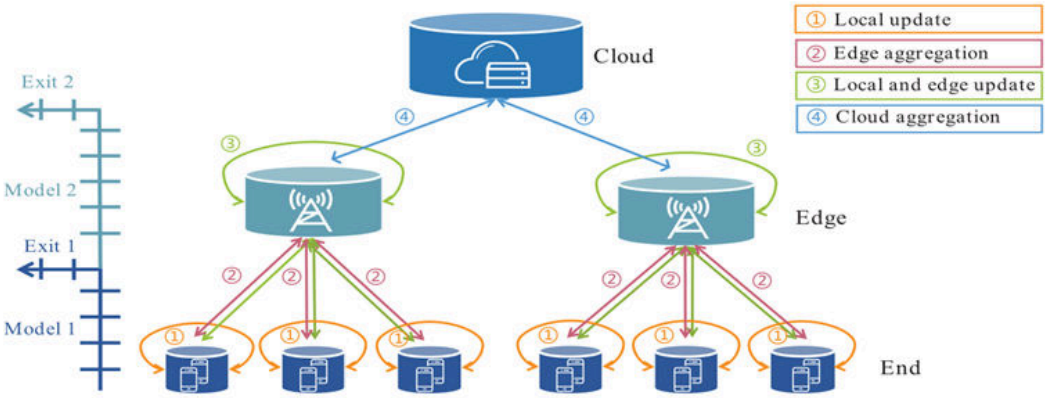

Figure 26: Distributed Federated Learning in an IoT-Edge-Cloud architecture [58]

6.1.4 Limitations of the EE techniques

EE is inherently applicable for classification problems, as we need a confidence score to determine if we want to exit early or continue processing the inference sample. Such a metric does not exist in regression tasks, where the outputs are numerical values. There has been little research to adapt it for regression, such as BERxiT [60], where an additional output is computed, called LTE (Learn to Exit). To this end, the network computes the scalar value as well as the confidence it has to its answer. Apart from that there is little adaptation of EE to regression problems. Furthermore, when building the model, access to the original data to train the added layers is needed, as well as knowledge of the architecture, placing of the exits in the optimal places/branches with the optimal threshold. Thus, this method cannot be applied out of the box in a model but has some fine tuning that needs to be made. Finally, in the worst case, its performance is deteriorated compared to simply offloading to sample the cloud, which, for some cases, is critical and thus, cannot be applied to every single case.

6.2 Knowledge Distillation techniques

Knowledge distillation (KD) is the process of using a bigger, trained network, also known as the teacher, who teaches the smaller model, or distills its knowledge, to the student. To understand how that is achieved, we need to understand an inherent problem with classification. An easy demonstration of said problem, let's consider the following example. We have an image classification problem, containing images of three animals, horse, zebra, and peacock. The conventional method would consist of humans labeling each image by hand and assigning a one hot encoding vector, either [1,0,0] , [0,1,0], [0,0,1], respectively. The images then are usually passed through a series of convolutional, Relu and Pooling layers that make up what is known as the feature extractor. These features are then processed by a neural network, with a SoftMax function, that classifies the image (Figure 27).

Funded by the European Union

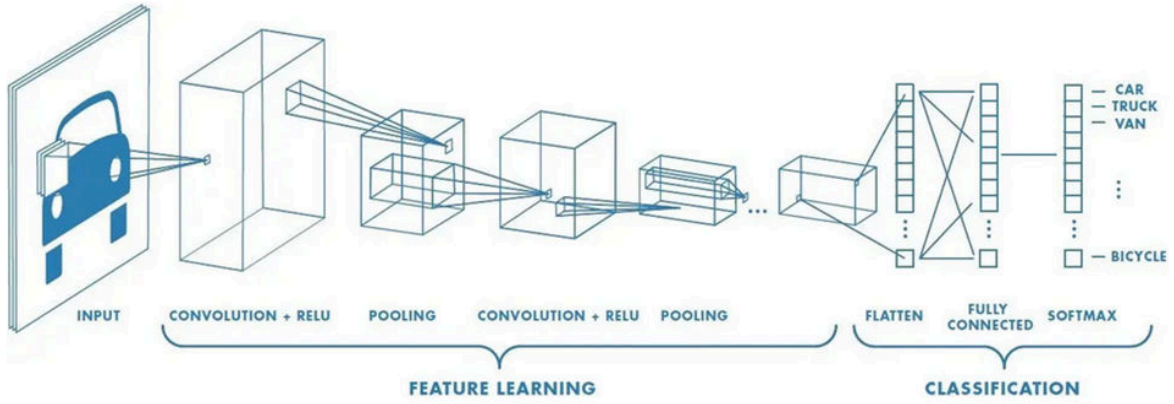

Figure 27: Feature learning and classification task of a Convolutional Neural Network (CNN)

The problem with this approach is that since the images are labeled based on class, the distance between the label vectors is the same between all classes. A peacock is as far away from a horse as a zebra, a fact that, intuitively, feels wrong, since a zebra looks very much like a horse. More importantly, it shares a lot of its features with a horse. The legs, the neck, the shape of the ears etc. Naturally, the feature extractor part of the CNN will pick up some of these, (or even some other patterns that don't really make sense to us humans). The important thing is that some of the features will be shared between the horse and the zebra, whereas much less with the peacock.

While training the network, usually in the later stages, when an image of a horse is given to the network, some of these common features will be picked up by the network. If the network is trained correctly, it will, to some extent, predict the zebra class. When backpropagation happens, the weights that resulted in our network inferring zebra will be "punished" or reduced as it was the wrong answer based on the hard labels provided by the human. The same will happen in the image of a zebra. This results in a sort of "tug of war" between the similar classes, in the overlapping features. For simplicity let's say this shared feature is "four legs". Each image of a horse will punish the connection between "four legs" and the label horse and vice versa. That is undesirable, as both have four legs. The shared features are contested as, by the nature of the labeling process, the classes are mutually exclusive. This problem can be combated by what is known as "soft labeling". Instead of a class being either 1 or 0, we have continuous values for each class. A comparison between the two is demonstrated in Figure 28.

Figure 28: Difference between hard and soft labels

If a network is trained using the soft labels, the shared features will be much less punished, as the distance between the actual and the predicted value will be much less. The next question that is raised is how we obtain these soft labels. That is where the teacher network

comes into play. We can use a big, trained network that we have confidence in. Before training the student, we pass the image through the teacher network and obtain a classification that is softer than the hand-picked hard labeling. If the outputs are not “soft” enough, we can modify the SoftMax function to what is known as “SoftMax temperature” [62]62]:

$$softmax(z)_i = e^{z_i/T} / \sum_j e^{z_j/T},$$

with the number T, being a hyperparameter that needs to be tuned appropriately during training, as depicted in Figure 29. The training process consists of feeding train images through the teacher network and using a combination of its outputs and the hard labels, to train the student network (Figure 30).

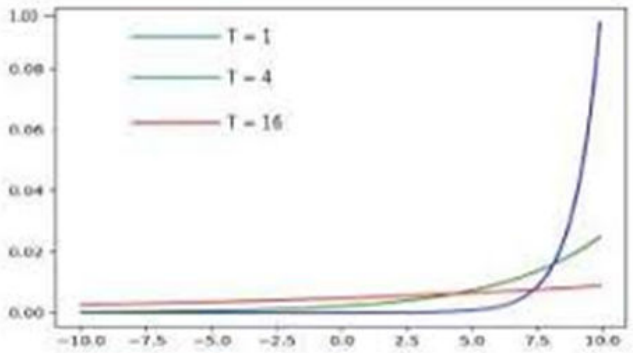

Figure 29: Softmax temperature with different values of T [63].

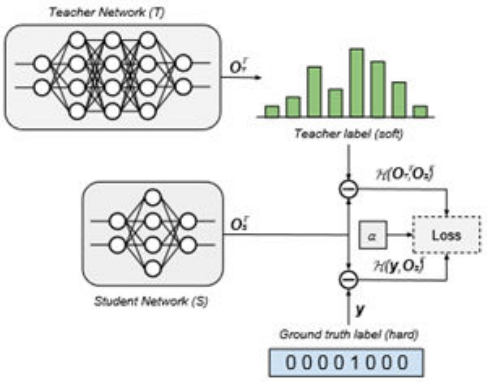

Figure 30: Combination of hard and soft labels, training the student network [61]

6.2.1 Data free Knowledge Distillation

One additional application of KD is its ability to train the student network without having access to the original data [64]. Borrowing a concept from Generative Adversarial Networks (GANs), the generator network generates synthetic data, which is then passed through both the teacher and the student [65]. The student’s goal is to imitate the output of the teacher, whereas the goal of the generator is to maximize this difference. Through this iterative process, the student learns the same patterns as the teacher, even if the original data is not available. Similarly, a noise vector can be used alongside the output of the generator, to explore patterns and weights that the generator might miss, by exploiting a possible local minimum, as shown in Figure 31.

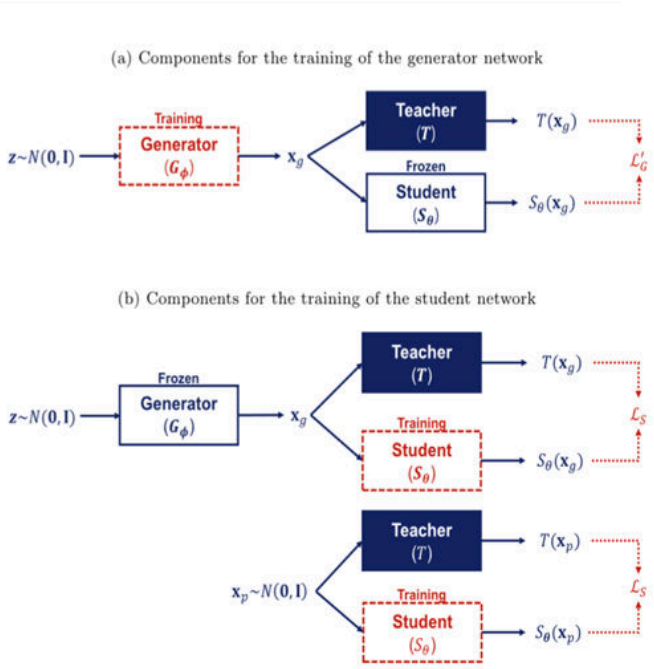

Figure 31: Data-free Knowledge Distillation model [64]

6.2.2 Applications of KD in Regression Tasks

6.2.2.1 Loss functions

As described above, KD is only applicable to classification problems. In contrast with EE, a lot of modifications have been made to facilitate regression problems. The simplest way to apply it to regression problems is to modify the loss function and include the difference between the student's and the teacher's output, as one would with the actual data. The simplest is to combine the ground truth loss with the teacher loss, weighted with a hyperparameter α., defining a loss function adapted for KD in regression [61]:

$$L_{reg} = 1/n \sum_{i=1}^{n} \alpha \left|\left|p_s - p_{gt}\right|\right|^2 + (1 - \alpha)\left|\left|p_s - p_T\right|\right|^2,$$

or use the teachers output to classify the outliers [66] in the dataset and neutralize their negative effect on the model [67]. If the teacher's output is far from the actual value, either the teacher is not trained correctly, or this datapoint is an outlier and thus must not be taken as seriously as the others, taking into account the following outlier loss function [67]:

$$L_{TOR} = \left|\left|R_s - t\right|\right|^2, \; if \; |t - R_t| \leq error_{outlier}$$

or

$$L_{TOR} = \sqrt{\left|\left|R_s - R_t\right|\right|}, \; if \; |t - R_t| \geq error_{outlier}$$

If the teacher result R_t has a difference bigger than the hyperparameter $error_{outlier}$ with the true value t, then we take the square root of the loss, instead of the usual Mean squared

error, mitigating some of its effect, thus making our model learn the general pattern, instead of overfitting on dangerous outliers.

6.2.2.2 Hint training

The problem with directly applying KD in regression tasks is that, by providing a single number, the student does not learn any patterns from the teacher, which is the case in classification problems. For the student to capture the "thought process" of the teacher, hint training compares the outputs of an intermediate layer of the two networks instead of the output neuron, as depicted in Figure 32. This can be done either by directly comparing their outputs [61], or by employing yet another method for GANs, this time the discriminator [65], that tries to tell the output of the teacher and the student apart. It must be noted that this method is not exclusive to regression tasks.

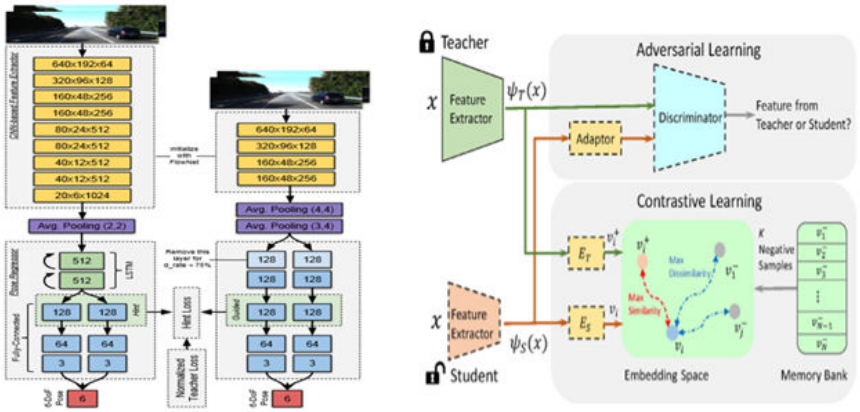

Figure 32: Hint training and feature extraction from Saputra [61] and Xu [65] respectively.

6.2.2.3 Reinforcement Learning

Furthermore, some cases of regression can be treated as an RL problem. Such is the case for stock prediction or currency prediction [68]. Instead of attempting to predict the exact value of the stock, we can train an agent that has one of several options such as buy, sell, exit, etc., as depicted in Figure 33. With this technique, we turn a regression problem to a classification one.

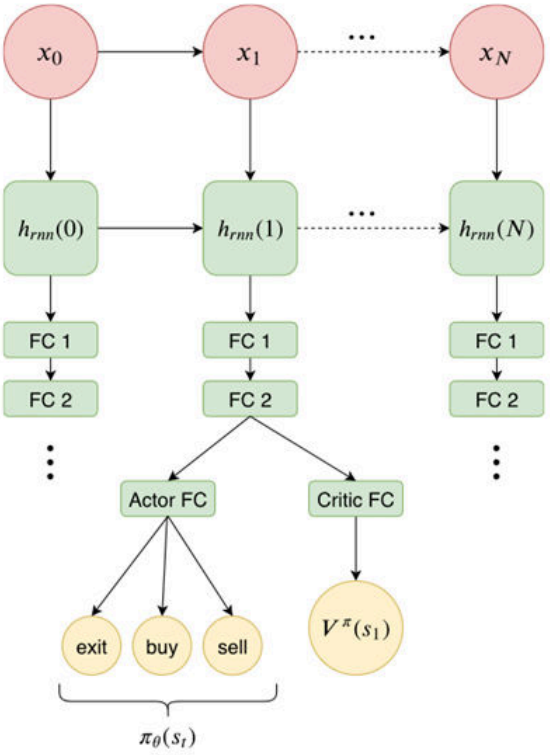

Figure 33: DRL agent decisions for a time series currency prediction and transformation of the regression to a classification problem [68]

6.2.3. Limitations of KD methods

Just as EE, the basic concept was developed on classification tasks and requires the dataset to train the students. These limitations can be overcome, but with some added complexity. The KD method is generally more flexible, but still requires some knowledge of neural network architecture to decide the student network size, train hyperparameters and any additional modifications.

6.3 Pruning, quantization, and related techniques for more lightweight ML training

This lightweight technique is seemingly simple, which adds a big advantage. It requires minimum knowledge from the user, to be applied, entailing setting specific weights of the network to zero. There are two types of pruning, structured and unstructured.

6.3.1 Structured and Unstructured Pruning

In unstructured pruning, we set the target weights to zero, based on our criterion of choice (details for the criterion are provided later in this document), as illustrated in Figure 34.

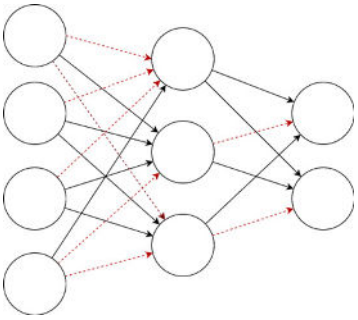

Figure 34: The Unstructured Pruning, illustrating the zeroing of specific target weights

On the other hand, in structured pruning, entire neurons are pruned and, as is natural, all the connections that start or end to that neuron, as depicted in Figure 35.

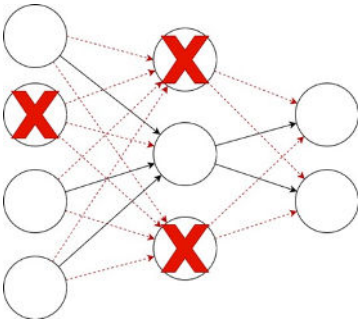

Figure 35: Structured Pruning, illustrating the zeroing of entire neurons

Intuitively, unstructured pruning seems more efficient, since it exhibits the flexibility to prune each weight individually, without them being bound to each other. When implemented, in order for unstructured pruning to achieve actual speed ups, the weight matrix has to be represented in a sparse form. Additionally, the operations have to be performed with a sparse algorithm. Without that, even if the matrix contains a lot of zeros, the floating-point operations will remain the same. With structured pruning there is no such issue, as every neuron is represented in a row of the weight matrix. Removing a neuron equals removing a row, which is very easy in practice.

However, pytorch does not support sparse matrix multiplication [69]. More specifically, it is mentioned that you cannot expect any inference speedups unless you use a custom sparse matrix algebra library to power your computation. `torch.sparse` is a library that is still a work in progress for now. Currently, the same number of operations are required before and after the pruning procedure; the difference is that after the pruning, a bunch of entries equal to zero in your tensors [70]. One must either create their own sparse multiplication library, or resort to structured pruning. It is obviously less efficient, as one's choices are more limited, and very easy to implement.

6.3.2 Pruning methods

Pruning weights do not happen at random. One has to define the strategy, or the metric based on which the less important weights are set to zero. It is important to keep in mind that the optimal method has been known to us since 1993 [71] and is called the optimal brain surgeon. It consists of expanding the error function, around the weights, as a taylor series by using:

$$\delta E = (dE/dt)^T \cdot \delta w + 1/2 \cdot (\delta w)^T \cdot H \cdot \delta w + O(|\delta w|^3)$$

We can safely ignore the term $O(|\delta w|^3)$. Additionally, we can assume that the term $(dE/dt)^T$ is close to zero, as our model is optimized. That leaves us with the second term that is non-negligible, also known as the Hessian matrix [72] [73]. One could use this information to compute not only the best target weights, but also the adjustment needed for the rest of the weights, to counterbalance that removal, using the following equations

$$L_q = (1/2) \cdot (w^2_{qq} / [H^{-1}]_{qq})$$

$$w = w_q / [H^{-1}]_{qq}) [H^{-1}] \cdot e_q$$

for the loss of the removal of each weight and the adjustment to the remainder of the weights respectively [71].

Unfortunately, the above process involves computing the Hessian Matrix, a $d \times d$ matrix, with d being the number of weights, as well as the second derivative of each element. For realistic problems, its computation is impossible. Thus, heuristic methods have been developed. The simplest ones consider either the magnitude of the weight, or the first derivative of the error about the weight [74]. Some more complicated methods try to approximate the Hessian matrix with an empirical Fisher information matrix, trying to simulate the diagonal of the Hessian, which seems to yield good results.

Finally, a new novel method created a DRL agent, trained the model to decide the optimal pruning rate, consistently outperformed the manually pruned network, relieving the agency of the expert, that comes hand in hand with some uncertainty that comes from the use of heuristic methods [75].

6.3.3 Pruning results

As a proof of concept, we present some preliminary results in Figure 36 for the pruning technique, using the nni Microsoft library on the CIFAR-10 dataset, also making use of the VGG pretrained model of the pytorch library. The results clearly indicate the advantage of applying a pruning method on a pre-trained model.

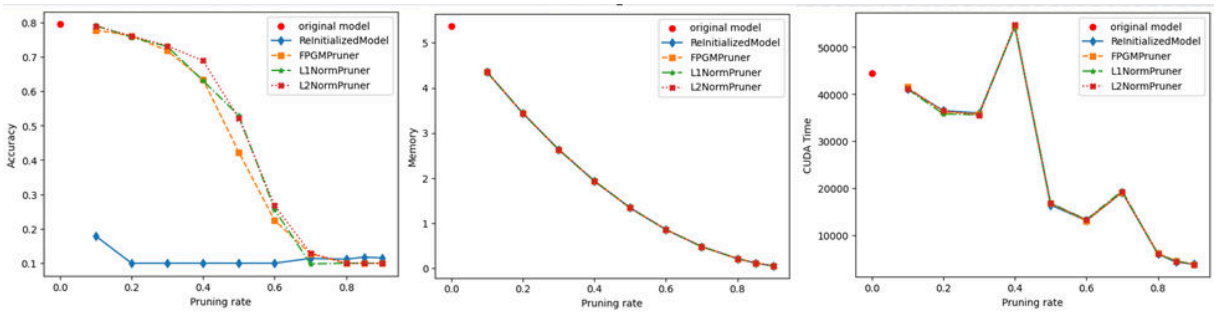

Figure 36: Our results, using the nni Library to prune the VGG model, applied on the CIFAR-10 dataset, including accuracy vs pruning rate (left panel), required memory vs pruning rate (middle panel) and CUDA time vs pruning rate (right panel).

6.3.4 Pruning Limitations

This approach is the one that can most easily be used out of the box, but has the most complicated implementation, as there is not enough open source support for every case possible. To this end, research activities will be carried out during the project specifically focusing on the pruning method, since it can be applied to all the TaRDIS use cases.

6.4 Discussion

Summarizing, we foresee that the aforementioned techniques for making a ML model more lightweight and energy-efficient will be probably implemented and demonstrated on all the Tardis use cases, since they involve the utilization of DNNs:

- Early-exit of inference: several exit points are included in the architecture of the DNN during its training and during the inference phase of the model. Thus, the output can be provided in the first model exit accompanied with a confidence/accuracy score. In case of an adequate confidence level, the model early-exits, saving resources (energy and computational, while also keeping the latency at ultra low levels). This method is preferably implemented in classification tasks and can be linked with the TID and ACT use case.
- Knowledge Distillation: it involves the process of intelligence extraction from a large DNN model (teacher) to a more lightweight one (student), saving energy and computational resources, while also keeping acceptable levels in the prediction accuracy of the DNN. Although this method can be applied to both classification and regression tasks, it exhibits enhanced performance when classification tasks are considered and thus, it will be linked with ACT, TID and possibly EDP use case.
- Pruning: it involves the nullification of several neuron weights in a DNN that do not have a significant impact in the performance of the model, accelerating the inference process and saving energy and computational resources. Since this method works well both for classification and regression tasks, it can be linked with all four use cases (TID, EDP, GMV, ACT).

In the framework of T5.3, these lightweight techniques have been identified, their input and output parameters have been investigated and tailored to the Tardis use cases, as well as the initial design of APIs has been carried out to associate these libraries with the Tardis toolbox. Furthermore, the pruning method that is considered relevant to all the use cases and therefore, the most critical has been developed, tested and validated against well-established datasets.

For the next period, the target of this task is to advance the development of all three techniques as python libraries and provide APIs in order to test them with open datasets, but also with data originating from the Tardis use cases. It should be noted that, ultimately, we aim to test these techniques in the distributed architecture of Tardis, validate and demonstrate their output in the use cases. For instance KD and pruning will be applied to each DRL model that resides in the smart home (EDP use case) to make the individual DDPG model energy-efficient, and early-exit will be combined with split learning in the ML model of each individual mobile device (TID use case).

Funded by the European Union

7 Conclusion

This report has documented the ongoing work and current results on the tasks dedicated to developing decentralized machine learning solutions. It has described the positioning of ML/AI tools in the TaRDIS framework, the identified ML modeling of the use cases and the advances on the development of:

i) a framework supporting AI/ML primitives,
ii) an AI-driven planning, deployment and orchestration framework, and
iii) lightweight, energy-efficient ML techniques.

The TaRDIS project task T5.1 will continue its activity aiming at expanding the list of supported FL algorithms, focusing on ML models for the use cases. Task T5.2 will continue its activity through further work on the Double Deep Q-Network algorithm, while focusing on additional challenges, advancing to the distributed version of the agent and implementing a FL approach. Task T5.3 will also continue its activity by advancing the development of all the three aforementioned techniques, i.e. early exit of inference, knowledge distillation and pruning, with the goal of rendering ML models more lightweight and energy-efficient. Given that real data originating from the TaRDIS use cases is currently not available, it remains for the next period to test the solutions developed within WP5 on real data sets, provided by the use cases. The results of these activities will be documented in the next deliverables (D5.2 and D5.3).

Funded by the European Union

8 ANNEX 1

In this annex, the mathematical framework of the HEMS algorithm is formulated. As noted, all the smart home variables are time-dependent. The energy stored in the ESS at time t is B_t and the storage dynamics can be described by:

$$B_{t+1} = B_t + w_c c_t + d_t / w_d,$$

where w_c and w_d are the charging and discharging energy coefficients (0, 1], while c_t and d_t are the charging and discharging power of the ESS respectively. Moreover, the energy level stored at the battery is limited between a minimum and a maximum value of energy storage capability and thus:

$$B_{min} \leq B_t \leq B_{max}$$

Another constraint that should be taken into consideration is the ESS charging and discharging rate limitations, i.e.:

$$0 \leq c_t \leq c_{max}$$
$$-d_{max} \leq d_t \leq 0,$$

whereas the simultaneous ESS charging and discharging can be avoided with the following limitation:

$$c_t \cdot d_t = 0$$

In order to maintain a comfortable environment inside the smart home, we assume that the temperature must be bounded within a certain temperature range:

$$T_{min} \leq T_t \leq T_{max}$$

and that the HVAC system with inverter in the smart home can provide an input power for the overall time slot continuously:

$$0 \leq h_t \leq h_{max}$$

The power balance equation can be used to satisfy the balance between the power drawn from the grid and the produced local power and the power that is requested by the smart home. To this end, the aggregated power supply should be equal to the served power demand:

$$u_t + r_t - d_t = c_t + h_t + g_t,$$

where u_t is the power drawn from the grid to balance the power deficit, r_t is the power generated by the renewable sources and specifically by the solar panels and g_t is the non-shiftable power demand that reflects the loads inside the smart home that need to be satisfied immediately. It should be noted that $u_t \leq 0$ reflects an energy surplus of the smart home that can be sold back to the grid (or sent to other smart homes through the community orchestrator). Otherwise, the smart home has an energy deficit and will purchase energy from the utility grid.

By using the aforementioned mathematical modeling, we can conceptualize the environment of a Smart Home as a Markov Decision Problem (MDP), considering our agent to be the HEM system [4]. In the MDP formulation, we inherently assume that the components of the environment at the next time slot depend only on their current value at the present time slot and not on past values. The environment state consists of seven kinds of information:

- renewable generation output r_t
- non-shiftable power demand g_t
- ESS energy level B_t
- Outdoor ambient temperature T_t^{out}
- Indoor temperature T_t^{in}
- Buying electricity price v_t
- time slot index in a day t' ($t' = mod(t, 24)$)

It should be noted that the selling electricity price p_t is typically linearly related to the buying electricity price v_t (e.g., $p_t = \delta v_t$). The state of the smart home environment can be described by the following tuple:

$$s_t = (r_t, g_t, B_t, T_t^{out}, T_t^{in}, v_t, t')$$

The action space of the HEMS agents is two-fold: (i) control the temperature to satisfy the users comfort range by manipulating the input power h_t of the HVAC system; (ii) minimize the energy consumption of the smart home by deciding between the charging c_t and discharging d_t of the ESS battery. We model this decision using one variable f_t : [−1, 1] → [$-d_{max}, c_{max}$] so when its value is positive the ESS system is charged $f_t \geq 0 \Rightarrow c_t = f_t$, whereas when the value is negative $f_t \leq 0 \Rightarrow d_t = f_t$. In order to minimize the energy cost we need to indirectly decide how much power will be drawn from the utility grid u_t by solving the power balance equation mentioned earlier after our agent makes a decision about h_t, c_t, d_t. To this end, the action of the agent is $a_t = (f_t, h_t)$.

Finally, the reward function of the HEMS agent can be described as follows (the reward includes three main terms that reflect the three objectives of the intelligent ML model):

- The penalty of deviation from the thermal comfort range in order for our agent to make proper adjustments to meet the user's demands.

$$R_{1,t}(s_t) = ([T_t^{in} - T_{max}]^+ + [T_{min} - T_t^{in}]^-)$$

- The depreciation of the ESS is given by (k is the depreciation coefficient of the ESS in $/kW that reflects the damage of the battery due to often charging or discharging):

$$R_{2,t}(s_t, a_t) = k(|c_t| + |d_t|)$$

- The cost of the energy consumption of the HVAC system based on the buying v_t and the selling electricity price p_t from/to power utility grid:

$$R_{3,t}(s_t, a_t) = |u_t|(v_t + p_t)/2 + |u_t|(p_t - v_t)/2$$

The final reward is given by:

$$R_{t+1} = R_{1,t}(s_t) - \beta(R_{2,t}(s_t, a_t) - R_{3,t}(s_t, a_t)),$$

where β is a weight coefficient.

In order to simulate temperature dynamics for the indoor temperature, we rely on the work of Zhang [6], Pilloni [7] and Constantopoulos [8] and use the following equation:

$$T_{t+1}^{in} = \varepsilon T_t^{in} + (1 - \varepsilon)(T_t^{out} - [\eta/A]h_t)$$

where $\varepsilon = 0.7$, $\eta = 2.5$ and $A = 0.14$ kW/F.

References

[1] Deliverable D2.2 - **Report on overall requirement analysis**, 2024, TaRDIS project, https://www.project-tardis.eu/wp-content/uploads/sites/101/2024/02/D2.2-V1.1-Final.pdf

[2] Deliverable D3.1 - **Report on the 1st iteration of the application model and APIs**, 2024, TaRDIS project, https://www.project-tardis.eu/wp-content/uploads/sites/101/2024/02/TaRDIS_D3.1-final.pdf

[3] Deliverable D4.1 - **Report on the desirable properties for analysis**, 2023, TaRDIS project, https://www.project-tardis.eu/wp-content/uploads/sites/101/2024/02/TaRDIS_D4.1.pdf

[4] J. O. Kephart and D. M. Chess, "The vision of autonomic computing", Computer, vol. 36, no. 1, pp. 41-50, 2003. Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, et al., "Engineering self-adaptive systems through feedback loops", Software engineering for self-adaptive systems, pp. 48-70, 2009.

[5] Deliverable D2.1 - **Report on the initial requirements analysis from co-design**, 2023, TaRDIS project, https://www.project-tardis.eu/wp-content/uploads/sites/101/2023/07/TaRDIS_Deliverable-2.1_v1.2.pdf

[6] Konečný, J., McMahan, H. B., Yu, F. X., Richtárik, P., Suresh, A. T., & Bacon, D. (2016). Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*.

[7] Hua, Y., Zhao, Z., Li, R., Chen, X., Liu, Z., & Zhang, H. (2019). Deep learning with long short-term memory for time series prediction. *IEEE Communications Magazine*, *57*(6), 114-119.

[8] Heating-RL_agent, retrieved January 22, 2024, https://github.com/Cernewein/heating-RL-agent/tree/master

[9] Yu, L., Xie, W., Xie, D., Zou, Y., Zhang, D., Sun, Z., ... & Jiang, T. (2019). Deep reinforcement learning for smart home energy management. *IEEE Internet of Things Journal*, *7*(4), 2751-2762.

[10] Qiu, C., Hu, Y., Chen, Y., & Zeng, B. (2019). Deep deterministic policy gradient (DDPG)-based energy harvesting wireless communications. *IEEE Internet of Things Journal*, *6*(5), 8577-8588.

[11] Zhang, D., Li, S., Sun, M., & O'Neill, Z. (2016). An optimal and learning-based demand response and home energy management system. *IEEE transactions on smart grid*, *7*(4), 1790-1801.

[12] Pilloni, V., Floris, A., Meloni, A., & Atzori, L. (2016). Smart home energy management including renewable sources: A QoE-driven approach. *IEEE Transactions on Smart Grid*, *9*(3), 2006-2018.

[13] Constantopoulos, P., Schweppe, F. C., & Larson, R. C. (1991). ESTIA: A real-time consumer control scheme for space conditioning usage under spot electricity pricing. *Computers & operations research*, *18*(8), 751-765.

[14] Data platform - Household data, retrieved January 22, 2024, https://data.open-power-system-data.org/household_data/

[15] Sandler, Mark, et al. "Mobilenetv2: Inverted residuals and linear bottlenecks." Proceedings of the IEEE conference on computer vision and pattern recognition. 2018.

[16] Maas, Andrew, et al. "Learning word vectors for sentiment analysis." Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies. 2011.

[17] Pradhan, Swadhin, et al. "Understanding and managing notifications." IEEE INFOCOM 2017-IEEE Conference on Computer Communications. IEEE, 2017.

[18] Ge, Fengpei, and Yonghong Yan. "Deep neural network based wake-up-word speech recognition with two-stage detection." 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2017.
[19] Hard, Andrew, et al. "Federated learning for mobile keyboard prediction." arXiv preprint arXiv:1811.03604 (2018).
[20] FLaaS: Federated Learning as a Service, retrieved January 23, 2024, https://github.com/FLaaSResearch
[21] Caldas, Francisco, and Cláudia Soares. "Machine Learning in Orbit Estimation: a Survey." *arXiv preprint arXiv:2207.08993* (2022)
[22] Reynolds, D. (2009). Gaussian Mixture Models. In: Li, S.Z., Jain, A. (eds) Encyclopedia of Biometrics. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-73003-5_196
[23] Chow, C.C., Wetterer, C.J., Baldwin, J. et al. Cislunar Orbit Determination Behavior: Processing Observations of Periodic Orbits with Gaussian Mixture Model Estimation Filters. J Astronaut Sci 69, 1477–1492 (2022). https://doi.org/10.1007/s40295-022-00347-7
[24] Blei, David M., Alp Kucukelbir, and Jon D. McAuliffe. "Variational inference: A review for statisticians." Journal of the American statistical Association 112.518 (2017): 859-877.
[25] G. Terejanu, P. Singla, T. Singh and P. D. Scott, "A novel Gaussian Sum Filter Method for accurate solution to the nonlinear filtering problem," 2008 11th International Conference on Information Fusion, Cologne, Germany, 2008, pp. 1-8.
[26] E. Variani, E. McDermott and G. Heigold, "A Gaussian Mixture Model layer jointly optimized with discriminative features within a Deep Neural Network architecture," 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), South Brisbane, QLD, Australia, 2015, pp. 4270-4274, doi: 10.1109/ICASSP.2015.7178776.
[27] Charu C. Aggarwal. 2018. Neural Networks and Deep Learning: A Textbook (1st. ed.). Springer Publishing Company, Incorporated.
[28] J. Pihlajasalo, H. Lepp¨akoski, S. Ali-L¨oytty, R. Pich´e, Improvement of GPS and BeiDou extended orbit predictions with CNNs, in: 2018 European Navigation Conference, ENC 2018, Institute of Electric and Electronics Engineers (IEEE), Gothenburg, Sweden, 2018, pp. 54–59.
[29] G. Curzi, D. Modenini, P. Tortora, Two-line-element propagation improvement and uncertainty estimation using recurrent neural networks, CEAS Space Journal 14 (2022) 197–204.
[30] J. F. San-Juan, I. Pérez, M. San-Martín, E. P. Vergara, Hybrid SGP4 orbit propagator, Acta Astronautica 137 (2017) 254–260.
[31] Cuomo, S., Di Cola, V.S., Giampaolo, F. *et al.* Scientific Machine Learning Through Physics–Informed Neural Networks: Where we are and What's Next. *J Sci Comput* **92**, 88 (2022). https://doi.org/10.1007/s10915-022-01939-z
[32] Raissi, Maziar, Paris Perdikaris, and George E. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations." Journal of Computational physics 378 (2019): 686-707.
[33] L. Ghilardi, A. Scorsoglio, R. Furfaro, Orbit determination with maneuver estimation in cislunar environment va physics informed neural networks, in: 2022 AAS/AIAA Astrodynamics Specialist Conference, Charlotte, USA, 2022, p. 13.
[34] Lagaris, Isaac E., Aristidis Likas, and Dimitrios I. Fotiadis. "Artificial neural networks for solving ordinary and partial differential equations." IEEE transactions on neural networks 9.5 (1998): 987-1000.
[35] Deliverable D7.1 - **Report on the expected improvements and quantification procedures**, 2023, TaRDIS project,

https://www.project-tardis.eu/wp-content/uploads/sites/101/2024/02/TaRDIS_D7.1-Report-on-the-expected-improvements-and-quantification-procedures_v1.1.pdf

[36] Siqi Liang, Jintao Huang, Junyuan Hong, Dun Zeng, Jiayu Zhou, Zenglin Xu, FedNoisy: Federated Noisy Label Learning Benchmark, 2023, https://arxiv.org/abs/2306.11650

[37] M. Savic, J. Atanasijevic, D. Jakovetic, N. Krejic, Tax evasion risk management using a Hybrid Unsupervised Outlier Detection method. Expert Syst. Appl. (2022)

[38] M. Savic et al., Deep Learning Anomaly Detection for Cellular IoT With Applications in Smart Logistics. IEEE Access (2021)

[39] Yu Gong et al., Variational Selective Autoencoder: Learning from Partially-Observed Heterogeneous Data, 2021, https://proceedings.mlr.press/v130/gong21a/gong21a.pdf

[40] L. Fodor, D. Jakovetic, D. Boberic Krsticev, S. Skrbic, A parallel ADMM-based convex clustering method. EURASIP J. Adv. Signal Process. (2022)

[41] A. Armacki, D. Bajovic, D. Jakovetic, S. Kar, Gradient Based Clustering, International Conference on Machine Learning (ICML), 2022

[42] Yu Zhang, Kanat Tangwongsan, Srikanta Tirthapura, Streaming k-Means Clustering with Fast Queries Yu Zhang, Kanat Tangwongsan, Srikanta Tirthapura, 2018, https://arxiv.org/pdf/1701.03826.pdf

[43] M. Popovic, M. Popovic, I. Kastelan, M. Djukic and S. Ghilezan, "A Simple Python Testbed for Federated Learning Algorithms," 2023 Zooming Innovation in Consumer Technologies Conference (ZINC), Novi Sad, Serbia, 2023, pp. 148-153, https://doi.org/10.1109/ZINC58345.2023.10173859

[44] M. Popovic, M. Popovic, I. Kastelan, M. Djukic, I. Basicevic, “A Federated Learning Algorithms Development Paradigm,” ECBS 2023 (to appear). arXiv:2310.05102 [cs.DC] https://doi.org/10.48550/arXiv.2310.05102

[45] Logistic Regression, retrieved January 23, 2024, https://colab.research.google.com/drive/1qmdfU8tzZ08D3O84qaD11Ffl9YuNUvlD

[46] D. J. Beutel, T. Topal, A. Mathur, et al., Flower: A friendly federated learning research framework, 2020. arXiv: 2007.14390 [cs.LG].

[47] C. T. Dinh, N. Tran, and J. Nguyen, “Personalized federated learning with moreau envelopes,” in Advances in Neural Information Processing Systems, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 21 394–21 405. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/f4f1f13c8289ac1b1ee0ff176b56fc60-Paper.pdf.

[48] Aleksandar Armacki, Dragana Bajovic, Dusan Jakovetic, Soummya Kar. A One-shot Framework for Distributed Clustered Learning in Heterogeneous Environments. IEEE Trans. Sig. Proc., to appear, 2024

[49] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y. Arcas, “Communication-Efficient Learning of Deep Networks from Decentralized Data,” in Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, A. Singh and J. Zhu, Eds., ser. Proceedings of Machine Learning Research, vol. 54, PMLR, 20–22 Apr 2017, pp. 1273–1282. [Online]. Available: https://proceedings.mlr.press/v54/mcmahan17a.html

[50] B. E. Woodworth, K. K. Patel, and N. Srebro, “Minibatch vs local sgd for heterogeneous distributed learning,” in Advances in Neural Information Processing Systems, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 6281–6292. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/45713f6ff2041d3fdfae927b82488db8-Paper.pdf.

[51] F. Hanzely and P. Richtárik, Federated learning of a mixture of global and local models, 2021. arXiv: 2002.05516 [cs.LG]

[52] Alberto Montresor and Mark Jelasity. PeerSim: A scalable P2P simulator. In Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09), pages 99–100. Seattle, WA, September 2009.

[53] Terry, J., Black, B., Grammel, N., Jayakumar, M., Hari, A., Sullivan, R., ... & Ravi, P. (2021). Pettingzoo: Gym for multi-agent reinforcement learning. *Advances in Neural Information Processing Systems*, *34*, 15032-15043.

[54] Baek, J. Y., Kaddoum, G., Garg, S., Kaur, K., & Gravel, V. (2019, April). Managing fog networks using reinforcement learning based load balancing algorithm. In *2019 IEEE Wireless Communications and Networking Conference (WCNC)* (pp. 1-7). IEEE.

[55] Shapley, Lloyd S. "Stochastic games." *Proceedings of the national academy of sciences* 39.10 (1953): 1095-1100.

[56] Teerapittayanon, S., McDanel, B., & Kung, H. T. (2016, December). Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd international conference on pattern recognition (ICPR)* (pp. 2464-2469). IEEE.

[57] X. Zhang, M. Hu, J. Xia, T. Wei, M. Chen, and S. Hu, "Efficient Federated Learning for Cloud-Based AIoT Applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 11, pp. 2211–2223, Nov. 2021, doi: 10.1109/TCAD.2020.3046665.

[58] Z. Zhong, W. Bao, J. Wang, X. Zhu, and X. Zhang, "FLEE: A Hierarchical Federated Learning Framework for Distributed Deep Neural Network over Cloud, Edge, and End Device," *ACM Trans Intell Syst Technol*, vol. 13, no. 5, Oct. 2022, doi: 10.1145/3514501.

[59] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Distributed Deep Neural Networks over the Cloud, the Edge and End Devices," in *Proceedings - International Conference on Distributed Computing Systems*, Institute of Electrical and Electronics Engineers Inc., Jul. 2017, pp. 328–339. doi: 10.1109/ICDCS.2017.226.

[60] Xin, J., Tang, R., Lee, J., Yu, Y., & Lin, J. (2020). DeeBERT: Dynamic early exiting for accelerating BERT inference. *arXiv preprint arXiv:2004.12993*.

[61] Saputra, M. R. U., De Gusmao, P. P., Almalioglu, Y., Markham, A., & Trigoni, N. (2019). Distilling knowledge from a deep pose regressor network. In *Proceedings of the IEEE/CVF international conference on computer vision* (pp. 263-272).

[62] Yang, J., Martinez, B., Bulat, A., & Tzimiropoulos, G. (2021, May). Knowledge distillation via softmax regression representation learning. International Conference on Learning Representations (ICLR).

[63] Takemura, Tatsuya et al. "Model Extraction Attacks against Recurrent Neural Networks." *ArXiv* abs/2002.00123 (2020): n. pag. https://www.researchgate.net/figure/Softmax-Function-with-Temperature_fig2_339015099

[64] Kang, M., & Kang, S. (2021). Data-free knowledge distillation in neural networks for regression. *Expert Systems with Applications*, *175*, 114813.

[65] Q. Xu, Z. Chen, M. Ragab, C. Wang, M. Wu, and X. Li, "Contrastive adversarial knowledge distillation for deep model compression in time-series regression tasks," *Neurocomputing*, vol. 485, pp. 242–251, May 2022, doi: 10.1016/j.neucom.2021.04.139.

[66] Takamoto, M., Morishita, Y., & Imaoka, H. (2020, August). An efficient method of training small models for regression problems with knowledge distillation. In *2020 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)* (pp. 67-72). IEEE.

[67] R. Kiani, A. Keshavarzi, and M. Bohlouli, "Detection of Thin Boundaries between Different Types of Anomalies in Outlier Detection Using Enhanced Neural Networks," *Applied Artificial Intelligence*, vol. 34, no. 5, pp. 345–377, Apr. 2020, doi: 10.1080/08839514.2020.1722933.

[68] A. Tsantekidis, N. Passalis, and A. Tefas, “Diversity-driven knowledge distillation for financial trading using Deep Reinforcement Learning,” *Neural Networks*, vol. 140, pp. 193–202, Aug. 2021, doi: 10.1016/j.neunet.2021.02.026.
[69] PyTorch, retrieved January 24, 2024, https://github.com/pytorch/pytorch
[70] Weight Pruning on BERT, retrieved January 24, 2024, https://discuss.pytorch.org/t/weight-pruning-on-bert/83429/2
[71] B. Hassibi’, D. G. Stork, and G. J. Ivolff, “0ptirna.l Brain Surgeon and General Xetlwork Pruning.”
[72] E. Frantar, S. P. Singh, E. Zurich, and D. Alistarh, “Optimal Brain Compression: A Framework for Accurate Post-Training Quantization and Pruning.” [Online]. Available: https://github.com/IST-DASLab/OBC.
[73] S. Pal Singh, E. Zurich, and D. Alistarh, “WoodFisher: Efficient Second-Order Approximation for Neural Network Compression.” [Online]. Available: https://github.com/IST-DASLab/WoodFisher.
[74] Blalock, D., Gonzalez Ortiz, J. J., Frankle, J., & Guttag, J. (2020). What is the state of neural network pruning?. *Proceedings of machine learning and systems*, *2*, 129-146.
[75] He, Y., Lin, J., Liu, Z., Wang, H., Li, L. J., & Han, S. (2018). Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European conference on computer vision (ECCV)* (pp. 784-800).

Funded by the European Union