Horizon 2020 Program (2014-2020)

A computing toolkit for building efficient autonomous applications leveraging humanistic intelligence (TEACHING)

D4.2: Report on integrated mockup of the AIaaS system[†]

Contractual Date of Delivery	31/08/2021
Actual Date of Delivery	20/09/2021
Deliverable Security Class	Public
Editor	*Claudio Gallicchio (UNIPI)*
Contributors	UNIPI: Vincenzo Lomonaco, Daniele Di Sarli, Antonio Carta, Rudy Semola, Andrea Cossu, Giacomo Carfì, Lorenzo Massagli, Federico Matteoni, Claudio Gallicchio HUA: Christos Sardianos, Iraklis Varlamis, Konstantinos Tserpes, Dimitrios Michail, Charalampos Davalas M: Salvatore Petroni CNR: Massimo Coppola, Pietro Cassarà ITML: Mina Marmpena
Quality Assurance	*Reviewer Jürgen Dobaj (TUG)*

[†] *The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871385.*

The TEACHING Consortium

University of Pisa (UNIPI)	Coordinator	Italy
Harokopio University of Athens (HUA)	Principal Contractor	Greece
Consiglio Nazionale delle Ricerche (CNR)	Principal Contractor	Italy
Graz University of Technology (TUG)	Principal Contractor	Austria
AVL List GmbH	Principal Contractor	Austria
Marelli Europe S.p.A.	Principal Contractor	Italy
Ideas & Motion	Principal Contractor	Italy
Thales Research & Technology	Principal Contractor	France
Information Technology for Market Leadership	Principal Contractor	Greece
Infineon Technologies AG	Principal Contractor	Germany

Document Revisions & Quality Assurance

Internal Reviewers

Jürgen Dobaj (TUG)

Revisions

Version	Date	By	Overview
1.0	20/09/2021	Claudio Gallicchio (UNIPI), Vincenzo Lomonaco (UNIPI), Antonio Carta (UNIPI), Christos Sardianos (HUA), Iraklis Varlamis (HUA)	Final, including Executive Summary, Introduction and Conclusions. Addressed comments.
0.9	14/09/2021	Jürgen Dobaj (TUG)	Comments on Draft
0.8	10/09/2021	Claudio Gallicchio (UNIPI)	Outline of Executive summary
0.7	03/09/2021	Vincenzo Lomonaco (UNIPI), Antonio Carta (UNIPI), Salvatore Petroni (M), Massimo Coppola (CNR)	Edits across Sections 2-7
0.6	31/08/2021	Claudio Gallicchio (UNIPI)	Comments to the second draft
0.5	30/08/2021	Salvatore Petroni (M), Vincenzo Lomonaco (UNIPI), Andrea Cossu (UNIPI), Federico Matteoni (UNIPI), Antonio Carta (UNIPI), Daniele Di Sarli (UNIPI), Lorenzo Massagli (UNIPI), Giacomo Carfì (UNIPI), Christos Sardianos (HUA), Iraklis Varlamis (HUA), Mina Marmpena (ITML), Massimo Coppola (CNR), Pietro Cassarà (CNR)	Second draft
0.4	23/08/2021	Claudio Gallicchio (UNIPI)	Comments to the first draft
0.3	06/08/2021	Salvatore Petroni (M),Vincenzo Lomonaco (UNIPI), Antonio Carta (UNIPI), Christos Sardianos (HUA), Iraklis Varlamis (HUA), Mina Marmpena (ITML)	First draft.
0.2	20/07/2021	Salvatore Petroni (M), Vincenzo Lomonaco (UNIPI), Mina Marmpena (ITML), Claudio Gallicchio (UNIPI)	Outline of the document
0.1	07/07/2021	Claudio Gallicchio (UNIPI)	ToC.

List of Tables

List of Figures

List of Abbreviations

AIaaS	Artifical Intelligence as a Service
AR	Application Runtime
CL	Continual Learning
CNN	Convolutional Neural Network
CPSoS	Cyber-Physical Systems of Systems
DL	Deep Learning
DMU	Decision Making Unit
EC	European Commission
ER	Emotion Recognition
ECG	Electrocardiography
EDA	Electrodermal Activity
ESN	Echo State Network
GRU	Gated Recurrent Unit
HAR	Human Activity Recognition
HSM	Human State Monitoring
HR	Heart Rate
IoT	Internet of Things
LM	Learning Module
LSTM	Long Short-Term Memory
LSTM-AE	Long Short-Term Memory Autoencoder
ML	Machine Learning
NN	Neural Network
RC	Reservoir Computing
RNN	Recurrent Neural Network
TF-LITE	TensorFlow Lite
WP	Work Package

Executive Summary

The Deliverable D4.2, entitled “Report on integrated mockup of the AIaaS system”, provides the first integrated mockup of the Artificial Intelligence as a Service (AIaaS) system, developed as part of the activities of TEACHING WP4. In particular, the deliverable includes the software demonstrator (AI-toolkit), which is made available through the TEACHING GitLab repository[1], and the corresponding report, which is given in this document.

The major goal of this document is to describe the AIaaS software toolkit, illustrating its updated architecture, the organization and API of the different software modules, as well as the integration mockup scripts and demos. Besides, to properly frame the content of this document within the scientific and methodological advancements carried out by WP4, we also give an update on the state-of-the-art and an overview of new preliminary results on relevant Artificial Intelligence methodologies.

This report is structured as follows. In Section 1 we introduce the scopes of this document and the relations with the other deliverables delivered at M20. Then, in Section 2 we present an updated state-of-the-art analysis that focuses on advancements in the core AI-based methodologies of interest. In Section 3 we present the architecture of the AIaaS system, while in Sections 4 and 5, we go in depth into the description of its fundamental pieces of software, respectively the *platform components* and the *learning modules*. Then, in Section 6 we describe the AIaaS mockup integration, showing how to proceed for the setup and the execution of the integration scripts. In the same section, we showcase several mockup use cases and describe two demo applications, which intend to demonstrate the potentiality of the developed system. In Section 7 we illustrate preliminary results on several research lines in AI which will be relevant for the further developments of the AIaaS during the rest of Y2 and in Y3. Finally, Section 8 concludes the document.

[1] https://teaching-gitlab.di.unipi.it/v.lomonaco/ai-toolkit

1 Introduction

The fundamental goal of the work performed in WP4 is to develop a distributed ***AI as a Service (AIaaS) software toolkit*** that enables ***human-driven adaptive applications in Cyber-Physical Systems of Systems (CPSoS)***.

The work is organized and performed in four tasks, whose description is briefly recalled below, including information from project replanning (in agreement with what reported in D7.2).
Task T4.1 "AI as a Service" is responsible for the development of the core methodological components of the TEACHING AIaaS, as well as of the design, implementation and integration of the software toolkit. The task started at M1, and it was planned to last until M30. After project replanning, the activities within T4.1 have been extended to last until M36.
Task T4.2 "AI for human monitoring" leverages the work in T4.1 to develop AI methodologies that are suitable for the recognition and characterization of the human state (physiological, emotional and cognitive) from streams of sensors' gathered information.
Task T4.3 "AI models for human-centric personalization" is responsible for the development of the self-adaptation and personalization functionalities of the AI modules in the AIaaS system, leveraging the human state information gathered from T4.2.
For both T4.2 and T4.3, the activities started at M7. While the original plan was for a duration until M36, after replanning both the tasks have been extended until M42. Moreover, their scopes have been extended to cope with anomaly detection towards avionics applications.
Task T4.4 "Privacy-aware AI models" focuses on the development of privacy-preserving methods to be bundled in the AIaaS system. The activities in this task were planned to start at M12 and end at M30. After replanning, for the task it is decided to have a duration from M15 until M42. In addition to that, the scope of the task has been extended to deal also with the relevant aspects of Dependable and Safe AI.

While the previous deliverable D4.1 was mostly dedicated to the preliminary design of the AIaaS system (Phase 1 of the project), this deliverable is intended to report the work performed during the first part of the core technology building activities (Phase 2 of the project). As such, the goals of this deliverable are the following:

- Provide an in-depth description of the refined architectural design of the AIaaS system;
- Illustrate the elementary software bricks of the AIaaS software toolkit, i.e., its platform components and its learning modules;
- Describe the AI-toolkit software repository, providing detailed information on the mockup integration and demo execution.

Moreover, with the idea of keeping the report on the scientific work carried out in WP4 up-to-date, D4.2 also includes an update on the relevant state-of-the-art and an overview of recent preliminary results.

The rest of this deliverable is structured as follows. In Section 2 we give an updated survey on the state-of-the-art in several AI-related methodologies of interest. Specifically, we give updates on the topics of Recurrent and Reservoir Computing (RC) Neural Networks, which have been previously identified (see Deliverable D4.1, Section 2) as fundamental AI building blocks for processing sequential forms of sensor-gathered data. This is complemented by advances in Federated, Continual and Reinforcement Learning methodologies, which are crucial to the development of the distributed AI learning services. The section also includes relevant information on privacy-preserving and anomaly detection AI algorithms. Moreover, in light of the great relevance played by the aspects of dependability and safety in relation to the introduction of AI methodologies within safety critical applications, we dedicate some space to

a summary of the literature in the field, sketching also the basic elements of a proposed methodology for safety compliance of Recurrent Neural Networks.
The updated state-of-the-art is followed, in Section 3, by the description of the refined AIaaS architecture, where we give an overview of the system (refreshing the rationale and the high-level requirements), as well as details on the formats adopted for data and metadata. In this section we also introduce the fundamental building blocks of the AIaaS architecture, namely the Platform Components and the Learning Modules, whose operation is reported in depth in Section 4 and Section 5, respectively. In Section 6, we report on the process of integration of the AIaaS mockup, introducing the software repository, the mockup setup and integration scripts, several use cases (covering the topics of sequence classification with Continual Learning, Dependability, Reinforcement and Federated Learning), as well as two demo applications for stress monitoring and autonomous driving personalization. After that, in Section 7 we intend to give the sense of the on-going scientific work in this WP, illustrating a number of preliminary results on fast learning in edge devices using TensorFlow-lite, adversarial robustness for sequence learning models, anomaly detection and Continual Learning (CL) with RC, as well as CL for Human State Monitoring. Finally, we draw our conclusions in Section 8.

Before that, in Section 1.1 we shortly recall the relations with the other deliverables.

1.1 Relationship with other deliverables

In compliance with its intended purpose within the scopes of the TEACHING project, this document (D4.2) describes the integrated mockup of the TEACHING AIaaS system.

D4.2 builds upon the previous WP4 deliverable D4.1, where we gave a preliminary version of the AIaaS design, which is now refined and complemented by a first description of the AIaaS mockup software components and of the mockup integration. At the same time, in D4.2 we provide an upgraded state-of-the-art analysis on relevant AI methodologies, and an overview of preliminary results on ongoing research.

This document is delivered within a group of related project deliverables, namely D1.2, D2.2, D3.2, D4.2, and D5.2 (listed in Table 1), all of which serve as a mean of verification for milestone MS2, entitled *First integrated setup with mock-up of the TEACHING platform*.

Table 1 Deliverable grouping for verification of TEACHING Milestone 2

D1.2	TEACHING CPSoS architecture and specifications
D2.2	Refined requirement specifications and preliminary release of the computing and communication platform
D3.2	Interim Report on Engineering Methods and Architecture Patterns of Dependable CPSoS
D4.2	Report on integrated mockup of the AIaaS system
D5.2	Preliminary use case deployment, implementation and integration report with related dataset release

The AIaaS system described in this document (D4.2) is framed within the context of the TEACHING CPSoS architectural concepts illustrated in D1.2, and relies on the High-Performance Computing and Communication Infrastructure (HPC2I), whose updated description is given in D2.2. Aspects related to dependable and safe AI are informed by the

work described in D3.2. Finally, the related body of work on use case deployment, implementation and integration is reported in D5.2.
The mapping of the viewpoints of the different WPs and related deliverables is depicted in Figure 1.

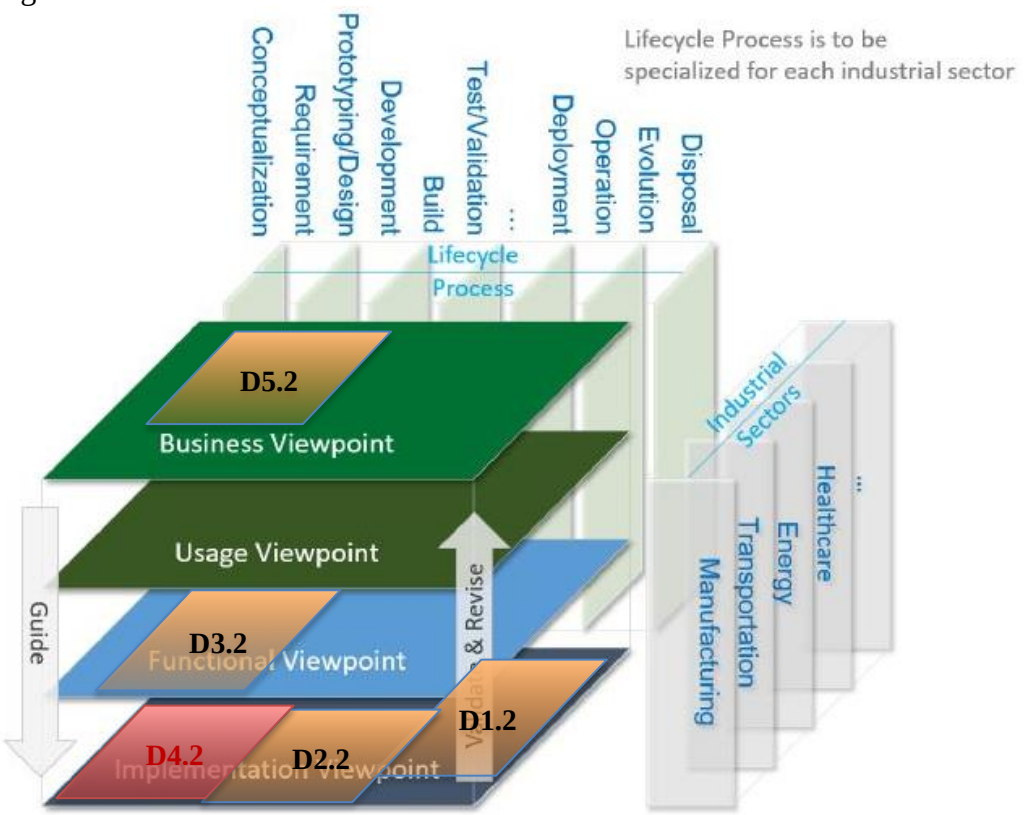


Figure 1 Depiction of the IIRA Viewpoints from[2] and mapping of TEACHING Deliverables focuses.

[2] https://iiot-world.com/industrial-iot/connected-industry/iic-industrial-iot-reference-architecture/

2 Updated State-of-the-art Analysis

This section aims at giving a refresher on the state-of-the-art methodologies that are of interest for the developments reported in this deliverable, touching a number of diverse topics. Notice that, to avoid cluttering and repetitions, whenever the fundamental literature on a topic has been already given in D4.1, here we focus on describing the advancements that resulted from the work within this work package.
Section 2.1 describes the recent advancements on Recurrent and RC neural networks, while Sections 2.2 and 2.3 give updates, respectively, on the fields of Federated Learning and Continual Learning. Section 2.4 introduces the fundamental concepts of Reinforcement Learning which are of interest and constitutes the basics for the human-centric personalization in autonomous vehicles applications. Section 2.5 introduces the basics of privacy-preserving approaches in ML. Section 2.6 dives into the topic of Dependability and Safety in AI applications for autonomous vehicles, also sketching the fundamental concepts of a proposed methodology for Functional Safety compliance of RNNs. Finally, Section 2.7 discusses the relevant state-of-the-art in the field of Anomaly Detection.

2.1 Recurrent and Reservoir Computing Neural Networks

The class of Recurrent Neural Networks (RNNs) [1] gives a flexible paradigm for learning with sequential forms of data. The key concept is that the neural network architecture includes a recurrent hidden layer that develops a contextual representation of the driving (possibly multi-dimensional) input signal. In this context, the Echo State Network (ESN) [2] [3] is the model of choice when computational efficiency of the training algorithms is of interest, as is the case of distributed learning on possibly low-powerful devices. ESNs are fundamentally based on the idea of exploiting the network activations from the point of view of a discrete-time dynamical system. For the sake of convenience, we recall that an ESN architecture comprises an input layer, a recurrent hidden layer (called the *reservoir*), and an output layer (called the *readout*).
In practice, the parameters of the recurrent hidden layer (i.e., of the reservoir) can be left untrained after proper initialization based on asymptotic stability conditions. A resulting striking advantage in comparison to conventional RNNs models is given by the extreme speed of training, as the learning problem formulation is much simplified (and typically boils down to a simple linear regression/classification). The idea of studying the evolution of the recurrent network as a dynamical system is not unique to ESNs but it is shared under the hat of the so-called Reservoir Computing (RC) paradigm [4].

RC in general and ESNs in particular offer a unique trade-off between complexity and accuracy, making them suitable for applications in learning tasks related to monitoring the human state conditions from sensor devices. For more information on RNNs, RC, and ESNs, the reader is referred to Section 2.1 of D4.1 (which also motivates in detail the adoption of the RC/ESN approach within the activities of WP4), or to the available literature surveys, e.g. [5] [6] [7].

To evaluate the performance of ESNs, and Deep ESNs [8], in comparison to the most common variants of RNNs over tasks of human state and activity recognition, we have performed a benchmark over a diverse set of networks and datasets, recently published in [9]. Our analysis comprised vanilla RNNs, Long Short-Term Memory networks (LSTMs) [9], Gated Recurrent Units (GRUs) [10], and their deep variations. The tasks considered in our analysis included a variety of cases in the area of Human State Monitoring (HSM), namely WESAD and ASCERTAIN, and Human Activity Recognition (HAR), namely HHAR, PAMAP2, and OPPORTUNITY. Full details on the datasets, as well as on the experimental conditions analysis

can be found in [9]. Here we limit ourselves to indicate that all the explored alternative learning models underwent an individual process of model selection in order to tune the respective relevant hyper-parameters, while keeping the number of trainable parameters comparable in all the cases. The major outcomes of our analysis are reported in Table 2, which shows the average accuracy (and std) achieved by the different models on the various tasks.

Table 2 Average accuracy and standard deviation on the test set. The averages and standard deviations are computed by retraining the models 5 times, each with a different random initialization of the weights. Reported from [9].

	WESAD		HHAR		PAMAP2		OPPORTUN.		ASCERTAIN	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std	Avg	Std
RNN	94.62	2.84	78.54	2.04	96.00	3.39	96.84	2.58	94.77	0.78
ESN	94.96	2.60	89.79	3.81	97.50	2.74	94.74	5.77	96.54	0.77
LSTM	95.48	1.17	92.71	2.72	96.50	1.22	93.08	2.88	94.63	0.00
GRU	98.13	1.16	98.54	0.83	98.50	2.00	96.84	2.58	94.63	0.00

The results clearly indicate that, despite their simplicity, ESNs are able to achieve a level of accuracy that is competitive with those models that require full adaptation of the parameters. Moreover, our results also point out that gated RNN architectures (especially GRUs) can be of particularly appealing in light of the high level of accuracy that can be achieved on HSM and HAR tasks. While in the current literature the GRU approach is fundamentally hampered by the high computational costs of the involved training algorithms, some of our recent research efforts have been devoted to find suitable hybrid RC-GRU methodologies that would be able to keep the advantages of both approaches. From a broader perspective, our analysis also points out that RNN in general can be a first choice for the class of tasks under consideration, in particular their deep and gated variants.

While the elementary characterization of RC methods is to leave untrained the recurrent hidden connections to reduce the training costs, a fundamental downside is that the developed temporal representations are achieved by a dynamical system that, in its settings, is agnostic with respect to the learning task on which it is applied. Hence, a relevant line of research in the field consists in looking for smart and cheap local learning algorithms, that are able to adapt some parts of the reservoir dynamics based on the task information [4]. Related to this aspect is the great interest for a specific dynamical regime of the recurrent hidden layer of an RNN/ESN, known as the ‘edge of criticality, or ‘the edge of chaos’ (EoC) [11]. This essentially represents the transition between stability and instability, where the computational properties of the recurrent layer are maximized. While the most widely known algorithm for tuning the reservoir dynamics, i.e. Intrinsic Plasticity [12], only indirectly leads to dynamical improvements of the reservoir system, in a recent paper we have introduced a novel training algorithm for ESNs which directly targets the proximity to the edge of EoC as an objective for optimization. The algorithm is named *Phase Transition Adaptation* (PTA) [13], and, as reported in Table 3, has been empirically demonstrated to sensibly improve the performance of conventional ESNs (and relevant architectural variants, including Simple Cycle Reservoirs – SCR [14]) on a set numerical benchmarks. See [13] for full details on the algorithm and the experimental analysis.

Table 3 NMSE (the lower the better) on the test set of classical RC benchmark datasets achieved by ESN, SCR and PTA. Best results arehighlighted in boldfont. Reported from [13].

Task	**ESN**	**SCR**	**PTA**
NLM	$9.30e-01_{(\pm 5.41e-02)}$	$9.80e-01_{(\pm 1.36e-02)}$	$\mathbf{6.76e-02}_{(\pm 7.47e-08)}$
NARMA	$1.61e-01_{(\pm 1.36e-02)}$	$1.77e-01_{(\pm 1.08e-02)}$	$\mathbf{1.33e-01}_{(\pm 3.04e-03)}$
MG	$9.31e-06_{(\pm 1.61e-07)}$	$7.14e-07_{(\pm 5.73e-08)}$	$\mathbf{5.90e-07}_{(\pm 6.35e-08)}$

2.2 Federated Learning

In a centralized setting, a Machine Learning algorithm can make use of all the available training data to produce a predictive model that best generalizes to unseen data. Unfortunately, a centralized setting is not always feasible. When the data comes from multiple independent devices, constraints such as network connectivity, bandwidth, and privacy preservation can make it impossible to aggregate the training data within a centralized location.

In a typical Federated Learning scenario [15], the aforementioned problem is tackled by letting each client produce a local Machine Learning (ML) model trained on just the locally available data. Then, instead of the raw data, it is the models that are transferred to a centralized location such as a server.

In the server, the models must be aggregated by strategy (e.g., averaging the weights) and then sent back to the clients if they need it for inference or further training. The critical point for an effective federation lies in the aggregation strategy, which ideally should produce a single compact model that incorporates all the knowledge from each client. However, due to the notorious difficulty in the interpretation of the weights of a neural network, it is not easy to give guarantees about the outcome of the aggregation.
One of the simplest techniques from the literature to enable Federated Learning for virtually all kinds of neural networks, including RNNs and ESNs, is that of training the models locally on each independent device, and then using on the server an aggregation strategy that is commonly known as *Federated Averaging* [16], illustrated in Figure 2.

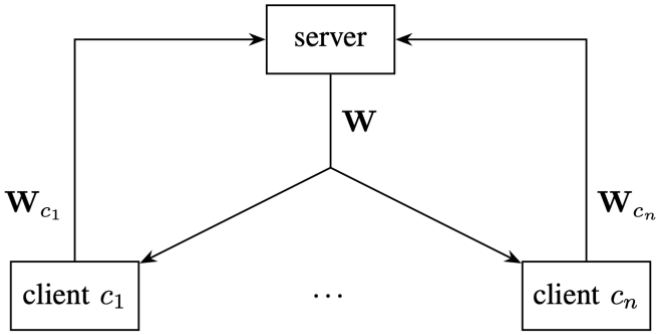


Figure 2 Federated Averaging Scheme. Each client c sends the local matrix Wc to the server. After the aggregation of the models is performed in the form of a weighted average, the server sends back the same matrix W to all clients. Taken from [17], to which the reader is referred for all details.

In the Federated Averaging strategy, the weights of each locally trained model are aggregated in the central server by an element-wise average, possibly weighted by the size of the local datasets. In the special case of ESNs, we can assume the scenario of a uniform configuration of the reservoir among all clients. In practice this means that the input-to-reservoir matrices and the reservoir-to-reservoir matrices will be identical in all clients. In this case, since the connections pointing to the reservoir are all untrained, Federated Averaging simply amounts to the transmission and averaging of the readout weights alone.

Averaging the readout weights is a straightforward technique that however does not give any strong guarantee about the performance of the aggregated model. We have proposed a novel federation method that can applied to ESN models, whose aggregation strategy guarantees optimal aggregated weights given the data and the reservoir. We refer to this strategy as "Incremental Federated Learning" [17], schematically depicted in Figure 3.

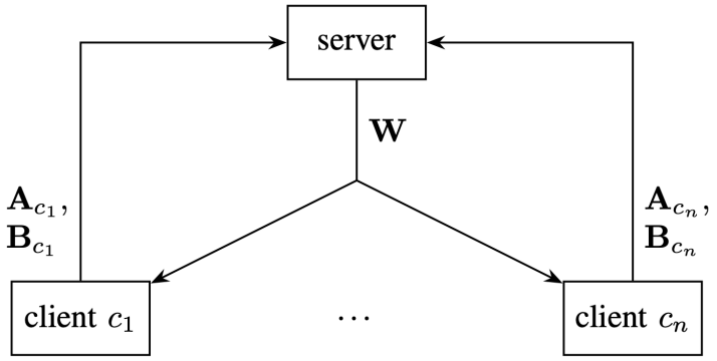


Figure 3 Incremental Federated Learning scheme. Each client sends the local matrices Ac and Bc to the server. After the matrices are aggregated and multiplied by the optimal readout weights, the server transmits W back to all clients. Taken from [17], to which the reader is referred for all details.

In Incremental Federated Learning, we exploit an algebraic decomposition of the typical readout training equation to produce two matrices that get transmitted to the server. These two are then recombined in the server, to produce readout weights that are mathematically equivalent to those that would have been computed if the data was locally available to the server. Full details are reported in the paper [17].

The advantages of the proposed approach are manifold. First, the long-proven characteristics of ESNs make it possible to train predictive models very efficiently, even directly on the edge. Second, the global model that is produced by aggregating the local models is optimal in the sense that no better equivalent model could have been produced by gathering all the training data within a centralized node. This point is illustrated by means of experimental analysis on two relevant datasets in the field of HSM and HAR, as shown respectively in Table 4 and Table 5, from which it can be seen that the performance of Incremental Federated ESNs matches the one of the centralized models and evidently outperforms the one achievable by Federated Averaging. Third, privacy constraints are preserved since the potentially sensitive training data is never transmitted over the network and remains confined within each local node.

Table 4 Training and Test accuracy on WESAD, achieved by Centralized model, Federated Averaging (FedAvg) and Incremental Federated (IncFed) Learning. Taken from [17], to which the reader is referred for full details.

Training subjects	Random baseline		Centralized ESN		FedAvg ESN		IncFed ESN	
	Training	Test	Training	Test	Training	Test	Training	Test
25%	25.30	24.59	93.14 ± .04	**65.96** ± .08	81.55 ± .09	63.42 ± .10	93.14 ± .04	**65.96** ± .08
50%	25.27	25.45	87.51 ± .06	**74.89** ± .09	77.63 ± .09	71.75 ± .11	87.51 ± .06	**74.89** ± .09
75%	25.09	24.32	84.45 ± .04	**76.56** ± .07	76.77 ± .11	74.69 ± .09	84.45 ± .04	**76.56** ± .07
100%	24.91	24.99	83.78 ± .06	**77.92** ± .06	76.57 ± .12	75.81 ± .10	83.78 ± .06	**77.92** ± .06

Table 5 Training and Test accuracy on HAR, achieved by Centralized model, Federated Averaging (FedAvg) and Incremental Federated (IncFed) Learning. Taken from [17], to which the reader is referred for full details.

Training subjects	Random baseline		Centralized ESN		FedAvg ESN		IncFed ESN	
	Training	Test	Training	Test	Training	Test	Training	Test
25%	16.65	15.77	92.73 ± .00	55.96 ± .01	99.17 ± .00	**60.38** ± .02	92.73 ± .00	55.96 ± .01
50%	16.82	17.13	93.60 ± .01	**70.28** ± .03	81.57 ± .07	66.99 ± .09	93.60 ± .01	**70.28** ± .03
75%	16.53	17.54	93.84 ± .02	**75.69** ± .05	78.52 ± .07	74.86 ± .06	93.84 ± .02	**75.69** ± .05
100%	16.75	16.99	93.64 ± .04	**80.19** ± .03	70.18 ± .07	74.84 ± .11	93.64 ± .04	**80.19** ± .03

2.3 Continual Learning

Learning continuously from non-stationary environments and ever-changing data streams is a complex challenge. Machine Learning algorithms and models today often assume an i.i.d. distribution of the underlying training data which should be representative of the (fixed) task to be solved. Hence, adaptation capabilities in many cases are only "s*imulated*" through an inefficient and expensive approach roughly based on three main steps: *i)* accumulate data; *ii)* re-train the prediction model from scratch on all the accumulated data; *iii)* re-deploy the prediction model. *Continual Learning* (CL), as a fast-growing field within the machine learning and deep learning community, aims at developing more efficient and scalable algorithms for incrementally acquiring new knowledge and skills and swiftly adapt to the ever-changing nature of the external world.

As already discussed in D4.1, continual learning may not only improve the effectiveness and sustainability of current AI solutions but also their robustness to catastrophic failures and enabling more privacy-preserving approach where data never leave the device on which they are collected / produce (and can be even deleted after training). However, CL poses challenges that are still difficult to overcome especially for gradient-based optimization algorithms such as neural networks that have been shown to suffer from *Catastrophic Forgetting* (CF) and the inability to learn from non-i.i.d. data streams.

In order to address these issues, several approaches have been devised and can conveniently be framed into a three-way fuzzy categorization: replay, regularization and architectural approaches [18]. However, these approaches are often very specific to the narrow scenario on which they are designed (e.g. class-incremental learning) and are difficult to port to even slightly settings. This impacts significantly on the ability to deploy these algorithms into the real world which are often tested on toy benchmarks and significantly constrained and artificial environments.

More recently and also through the TEACHING efforts [19], the research community has started to:

1. Extend the scope of current continual learning algorithms to different or more general scenarios other than the more common *Task-Incremental*, *Class-Incremental* and *Domain-Incremental* ones;
2. Apply current CL techniques to a broader range of applications such as *Human State Monitoring* (HSM);

As for 1., state-of-the-art continual learning approaches have mostly been tested on tasks where data points are not temporally coherent and can be processed in isolation. However, this is rarely the case for realistic use-cases where the temporal correlation often implies a semantic in the data. *Continual Sequence Learning* and *Classification* is of uttermost importance for many applications including the ones supported by TEACHING.

In [20], an initial study on how CL learning strategies work on different sequence learning tasks and datasets for sequence classification is reported. To the best of our knowledge this is one of the first attempts to provide a comprehensive empirical evaluation of continual learning with deep recurrent neural networks. This work opened the path to the analysis of continual learning with randomized neural networks for sequence modeling and in particular echo state networks (more on this in Section 7).

A significantly important application for TEACHING is human state monitoring: being able to understand the state of the human in *Cyber-Physical Systems of Systems* is crucial to adapt system properties and behavior in order to reduce stress, boredom, errors and maximize performance. However, this is not an obvious task if the environment, humans in the loops, objectives change over time as the system has to be constantly re-trained with high frequency.

Continual learning has been only recently investigated in this important area of application that can still be considered as a sequence learning and classification problem. In Section 7 more details about first experiments carried out on this topic are reported.

2.4 Reinforcement Learning

The task of autonomous driving can be approached with two different approaches [21]. The first assumes a ***modular pipeline*** that combines sensing and acting modules and machine learning modules in order to understand the vehicle environment and decide on the actions to be taken (e.g., on the driving profile to be selected). The actions are taken by the acting modules and control the vehicle behaviour based on predefined setups. The second assumes an ***end-to-end learning*** setup in which a neural network takes as input data from sensors and decides on the commands to be sent to the actuators (i.e., brake, gas, steer). The benefit from the former approach is on the ability to separately evaluate and control the behaviour of each module, and also on the ability to add more sensors or modules in order to further improve the vehicle navigation. Developing each module separately makes the overall task much easier as each of the sub-tasks can independently be solved by popular approaches for each task. In contrast, the latter approach demands a complete retraining of the ML model, when a new sensor is added or an actuator is replaced and any outlying behaviour can hardly be traced and reasoned.

In order to train an end-to-end learning approach or a machine learning module in the modular pipeline, which decides the vehicle actions based on the sensors' input, we can either employ imitation learning [22] [23] or reinforcement learning [24] techniques. In the case of imitation learning, the systems learn to operate the vehicle by monitoring how human drivers behave and

imitates their reactions [25]. The main advantage of this technique is the limited effort needed for training, since there is no need to define or describe the driving model, but simply to provide the model with training data. Its main restriction is the limited ability to generalise, since when the vehicle is trained in a specific environment (e.g., in a highway), we cannot be sure of its behaviour on a different environment (e.g., in a local road).

Reinforcement learning requires a more detailed description of the task, and the system will be able to learn after a long number of trial-and-error repetitions. A limitation is that the system cannot be trained in real conditions (e.g., on an actual car) since it learns through its errors, which in a real setup could be catastrophic in the case of an error that leads to an accident. For this reason, RL is usually performed on simulated environments, before we can consider the RL model ready to be deployed. These features make RL more appropriate for the modular approach, in which the final decision of the RL model is fed to an acting module (e.g., a driving mode selector) and not directly on a vehicle actuator, e.g., on the gas or brake.

Having all the above in mind, the driving personalisation module in TEACHING has been modelled using the modular pipeline approach. At the heart of this approach is the Reinforcement Learning module, which takes as input the streams of vehicle or driver sensors and decides on the driving mode that will be employed at each moment. Reinforcement learning has been used in the context of ADAS, in order to personalise the driving experience [26] and has been used in HCI, HRI, and CPSoS in general, for personalising the user experience [27] [28] [29].

The RL technique is based on the process by which an agent is trained by giving him the state of the environment he is called upon to solve and some reward policy. The agent tries different actions and after evaluating the action receives a reward. The same process is repeated until the agent solves the problem. An integral part of the RL solution is the definition of the RL task, as depicted in Figure 4.

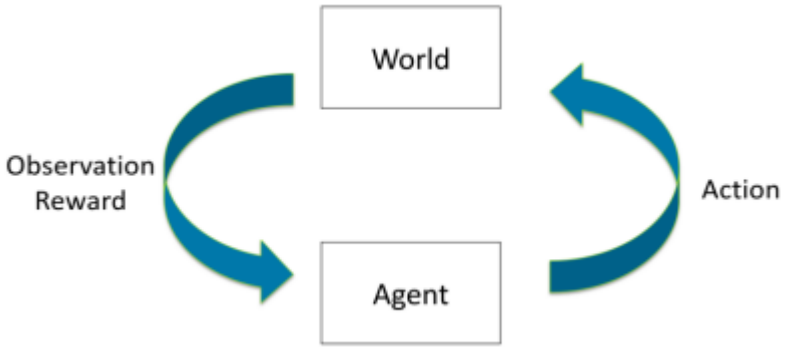


Figure 4 The basic idea and elements involved in a reinforcement learning model.

RL algorithms are classified into two main categories depending on whether or not they are model-based (Model-Based, Model-Free). This distinction refers to whether an agent, during training or performing a move, uses environmental predictions for his move, such as the rules of a game. In the project we dealt with the Model-Free category, as the problem of personalization is a complex problem and it is almost impossible to model driving and its personalization.

The next division of model-free approaches is in Policy Optimization and Q-Learning which are both based on the Markov Decision Process and their algorithms have several similarities

in their structure and performance. However, they differ radically in how they approach the choice of the next move by the agent. The goal of Q-Learning is to teach the agent to learn a deterministic solution that comes from a set of possible moves that will always choose the one with the highest reward (value based). In Policy Optimization, the solution can be stochastic and the agent aims to learn the best way in which a state of the environment is connected, with the next move (policy based). The Policy Gradient, A2C / A3C (Advantage Actor-Critic), PPO and TRPO algorithms are briefly classified in Policy Optimization and the DQN, C51, QR-DQN and HER algorithms in Q-Learning.

The Advantage Actor Critic algorithm is often found in two variants, A3C (Deepmind Async method) and A2C (OpenAI Sync method). A3C refers to its initial implementation by [30] and has to do with the parallel execution of multiple asynchronous instances of the agent, while A2C has to do with a later study by [31] which has shown that asynchronous execution does not significantly contribute to performance but can often reduce the efficiency of sample collection.

2.5 Privacy-preserving Learning

Machine learning models are often trained on sensitive data, such as health parameters, daily routes, or other identifying information. It is of paramount importance to guarantee the user's privacy in these scenarios, and for this reason, it must be ensured that the model does not leak private information about the user's data.

The most popular methods to enable privacy-aware training of machine learning models are based on the notion of differential privacy. The idea behind it is that a model is considered private if it is not possible to determine if a specific sample was present in its training set or not. The differential privacy guarantees depend on a privacy budget $< \epsilon, \delta >$, which can be used to control how private the model should be.

In TEACHING, privacy-aware training is a fundamental concern. For example, in the automotive setting, no data should leave the car, and the resulting model should guarantee the user’s privacy.

The main machine learning models used in TEACHING are ESNs and RNNs. Therefore, privacy-aware training is based on the differentially private SGD [32], a privacy-aware training algorithm designed for deep learning models. Currently, the literature on privacy-aware training is focused on feedforward and convolutional models, while recurrent models are under-studied. Our work in TEACHING could provide practitioners with useful guidelines to improve privacy-aware training of recurrent networks.

2.6 Dependable and Safe AI

In the previous deliverable D4.1 we introduced a number of metrics to ensure the dependability of NNs in order to use them in safety critical applications. Considering that TEACHING project focuses on sequential data we need to study how to ensure dependability of RNNs to be used in safety critical applications, such as in the automotive field. Automotive applications require adherence to Functional Safety for road vehicles [33] because without safety assurance the system can cause physical injuries or damage to the health of persons. If RNNs used to perform

human state monitoring do a mistake in the prediction of the psychological state of the driver, the driving of the vehicle could be entrusted to the driver even if the latter is not able to drive the vehicle. This scenario can cause an accident with negative impacts on the safety of the driver, passengers, and people in close proximity to the vehicle.

For this reason, the prediction of the psychological state of the driver is a safety-related task, which means that the system needs to be designed following Functional Safety guidelines. A feature of RNNs and, in general, of all the Neural Networks, is that the core of these software elements is not easily interpretable by humans and can be considered as a black box. Even with a completely deterministic model, the RNN's computations are complex and difficult to understand under all the possible scenarios. As a consequence, the model may fail in unpredictable ways. Keeping this consideration in mind, we can say that the most popular Functional Safety standard for Road Vehicles, the ISO 26262 [33] cannot be applied to design and test Neural Networks because it refers to the development of traditional software, where the behaviour is simpler and explicitly defined by the programmer. A new standard that can overcomes this problem is the ISO 21448 (version prepared for DIS), also known as SOTIF [34], a standard born to address the challenges introduced by autonomous driving systems with automation levels from 1 to 5.

SOTIF analyzes the possible behaviours of a function of the system (or the possible behaviours of a single element) that differ from the intended/desired behavior to verify if there are possible known scenarios that can be exploited to harm people. Furthermore, SOTIF tries to find out also possible unknown scenarios that can harm people. Once scenarios of potential risk for people's health have been discovered, SOTIF provides guidelines to mitigate the risk to an acceptable level. When all the possible safety risks have been mitigated to an acceptable value, the function can be released. One key requirement posed by SOTIF on each algorithm, component, and, in general, to the entire system is robustness. Robustness is usually understood as the ability of a system to react to adverse events, such as noise injection to the system inputs. However, robustness is not sufficient to consider a RNN safety. We need of some indicators capable of measuring the "safeness" of the RNN.

To address this challenge, we submitted a paper where we propose a methodology to reach Functional Safety compliance of RNNs [35]. First of all, we verify the robustness of RNNs with respect to inputs perturbations, such as those generated by systematic errors in the sensors data acquisition, environmental conditions, or adversarial perturbations [36]. To ensure the safety, RNNs must be robust to all these different noise sources. We propose a methodology that uses the robustness of the model, computed with state-of-the-art methods such as POPQORN [37], with respect to a range of accuracy values. By themselves, this robustness analysis of the RNN does not provide sufficient information about the safety of the RNN. For this reason, our methodology also provides a method to evaluate how often we are potentially unsafe through the use of Safety Performance Indicators (SPIs) [38] that count the number of unsafe occurrences. Depending on the specific needs of the application, a set of appropriate SPIs can be defined, along with the target values to be reached. Finally, a number of test scenarios must be performed and evaluated for each SPI. In the following we explain the phases to make a RNN compliant with Functional Safety.

2.6.1 Determination of RNN Adversarial Robustness by Inputs Perturbation

The first phase relies on the evaluation of robustness using POPQORN [37], where we measure how much noise can be injected into the input samples before the RNN's accuracy decreases

below a predefined threshold. More formally, given an input sequence X_0 we can add noise and move towards the input X_0' that is at a distance Δ from X_0 (Figure 5). Let us focus on a sequence classification task: for small values of Δ, the correct output of the network should be the same class of the original sequence, therefore hinting at a robustness of the RNN to small perturbations.

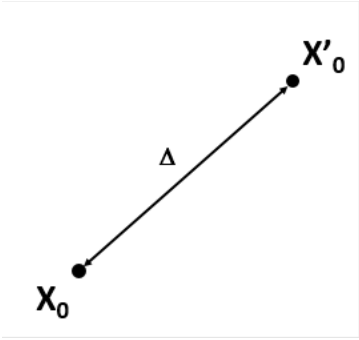


Figure 5 Sample X_0 and perturbed sample X_0' at a distance Δ.

Now, let us provide a sample X_0 as input to a properly trained RNN and suppose that the RNN will provide as output a correct prediction Y_0. We can identify a region of space around the sample X_0 such that all input samples $X_0' = X_0 + \Delta, \Delta < d$ inside the region will be correctly classified as the class Y_0 (Figure 6), with a determined accuracy (e.g., accuracy greater than 95%).

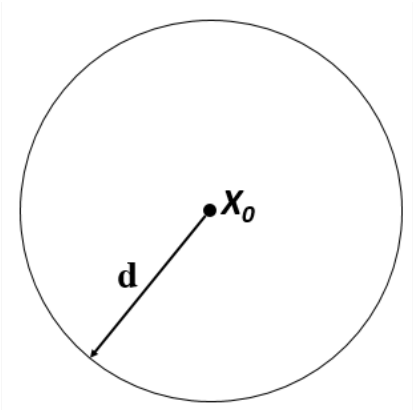


Figure 6 Input space around X_0 providing a correct prediction with a determined accuracy.

The goal of this first analysis is to determine three input spaces around X_0 where all the samples inside each region are correctly classified with a probability above a specified minimum value (e.g. 0.95, 0.8, 0.5). As a result, we obtain a measure of the local robustness of the RNN around the original input X_0, which are the three concentric hyperspheres corresponding to the different accuracy levels (Figure 7).

To accomplish this goal, we define three different RNN output threshold values: a_1, a_2, a_3 with the following relations: $a_3 < a_2 < a_1$. After, we shall apply different perturbations to the input sample X_0 to generate three different spaces as follows (Figure 7):

- S_1, space of the perturbed inputs with distance $d \leq d_1$ from X_0;
- S_2, space of the perturbed inputs with distance $d_1 < d \leq d_2$ from X_0;
- S_3, space of the perturbed inputs with distance $d_2 < d \leq d_3$ from X_0.

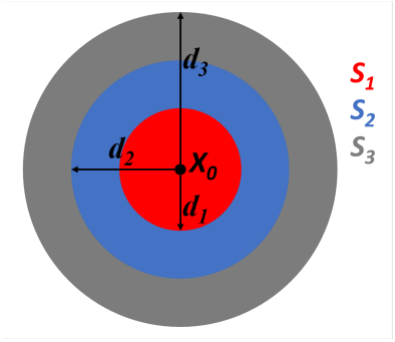


Figure 7 Input samples with distance $d \leq d_1$, $d_1 < d \leq d_2$ and $d_2 < d \leq d_3$ from X_0 belonging respectively to spaces: red, blue and grey.

Indicating with s a generic sample, we want to determine the values d_1, d_2 and d_3 such that:

- providing as RNN input the samples $s \in S_1$, the corresponding outputs have an accuracy $a \geq a_1$;
- providing as RNN input the samples $s \in S_2$, the corresponding outputs have an accuracy $a_2 \leq a < a_1$;
- providing as RNN input the samples $s \in S_3$, the corresponding outputs have an accuracy $a_3 \leq a < a_2$.

The process to determine the values d_1, d_2 and d_3 must be repeated over an adequate number of samples in order to have a more robust evaluation of values d_1, d_2 and d_3. To accomplish this, a number N of input samples must be considered, resulting in N different sets of values of values d_1, d_2 and d_3, one set for each sample:

$d_1^0 \quad d_2^0 \quad d_3^0 \quad$ for X_0
$d_1^1 \quad d_2^1 \quad d_3^1 \quad$ for X_1
$d_1^2 \quad d_2^2 \quad d_3^2 \quad$ for X_2
…
$d_1^N \quad d_2^N \quad d_3^N \quad$ for X_N.

We can compute average/max statistics from these values to determine the RNN's robustness as indicated below:

$d_1 = mean\{d_1^0, d_1^1, d_1^2, \dots, d_1^N\}$
$d_2 = mean\{d_2^0, d_2^1, d_2^2, \dots, d_2^N\}$
$d_3 = mean\{d_3^0, d_3^1, d_3^2, \dots, d_3^N\}$.

Notice that we do not give here any minimal values for d_1, d_2 and d_3 since these will depend on the specific application and the chosen samples. The values should be used to compare between different models.

2.6.2 Design of Safety Measures for Plausibility Checks

The second phase is aimed to evaluate the results achieved from the previous phase to establish which are the more appropriate safety measures that shall be applied. In this context safety measures can consist in a plausibility check to verify the information provided by the RNN. The plausibility check is provided by using one or more parallel redundant systems that can be

different algorithms that exploit the same inputs of the main system based on the RNN or can be systems with algorithms and sensors of different technology (for example radar or lidar) and so on. After the RNN based system and the redundant systems have computed their input data, there is a third system, the comparison system, that is responsible to perform a comparison of the results achieved and shall decide if the output of the RNN based system can be plausible or not (Figure 8).

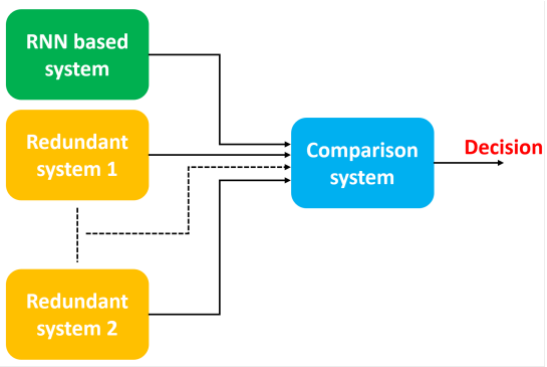


Figure 8 The comparison system performs the plausibility check among the RNN output and the output of one or more redundant systems and makes a decision.

In the case that the output of the RNN based system is judged not plausible by the comparison system and there is no condition for making a decision, the comparison system shall bring the system (and the vehicle) to the proper safe state depending on the current situation. However, to decide which is the more appropriate safety measure, the size of the spaces S_1, S_2, and S_3 determined during the evaluation of the adversarial robustness of the RNN shall be taken into account. For example, in the case that S_1 is much bigger than S_2 and S_3 can be considered negligible, we can consider the RNN quite reliable and so the safety measure can be constituted by an unsophisticated redundant system. A different case, for instance, involves a configuration where S_1 is bigger than S_2 but it is not predominant compared to this latter and S_3 can not be considered negligible; in this situation we shall design a more robust safety measure consisting of more redundant systems. The safety measure, comprising one or more systems, as well as the system based on the RNN and the comparison system, shall comply with specific time constraints provided by the application requirements. Safety critical applications shall operate in real time and so time constraints shall be considered during the design of the RNN based system, the redundant system (or systems) and the comparison system.

2.6.3 Safety Validation: Determination of SPIs and Test Length

The third phase consists in validating the achieved overall system, which includes the RNN based system and the safety measures adopted (redundant system or systems).
The SOTIF provides strategies to verify and validate the system, determining whether the risk associated to the function is reasonable (and so acceptable) or not. The verification step consists in the testing of the function against the known hazardous scenarios, that are those situations in which the function does not behave as expected causing a potential harm for involved people. The goal of the tests is to demonstrate that the potentially hazardous scenarios have been properly managed, and the associated risk previously discovered can now be considered reasonable and so acceptable.

After the verification, the functionality of the system shall be validated. The validation consists in the execution of tests to discover if there are unknown scenarios that can be potentially hazardous and so cause harm for involved people. To discover such unknown scenarios a series of tests are performed; such tests are aimed to observe the behavior of the function in as many real-life scenarios as possible: if the behavior deviates from the desired one and a potentially hazardous situation causing an unreasonable risk for the safety of people is found, some (or additional) safety measures shall be planned and developed to reduce the risk at an acceptable value. To measure the performance of a functionality, the SOTIF suggests KPIs as metrics; the KPIs are aimed to evaluate the performance of the functionality, that is the quality of the functionality. But from a Functional Safety point of view, we are interested not exactly to the general quality of the functionality, that is how well the functionality performs, but we are interested to evaluate how often the functionality is potentially unsafe. For this reason, it is better to use indicators that give a measure of the safety performance, the so called SPIs (Safety Performance Indicators) [38].

The SPIs give a measure about the dangerousness of the functionality (including the RNN) being tested, by telling us (for example) if there are dangerous misbehaviors, dangerous gaps in the considered ODD (Operational Design Domain), dangerous gaps in fault responses, dangerous defects in requirements, design, etc. In other words, an SPI gives a measure of the arrival rate of adverse events. SPIs shall be determined at different abstraction levels; so, we have SPIs for the overall functionality (or system), SPIs for the immediate sub-functionalities (or sub-systems) up to SPI for the atomic elements such as the RNN based algorithm, sensors, etc. Once the SPIs have been defined, for each of them you shall define the target value, a threshold value that each SPI shall not exceed to consider the safety related risk associated to the functionality acceptable. This threshold value indicates the risk budget that you do not want to overcome when your tests ended.

Before starting of the testing phase, a suitable test length shall be determined [34]. The test length expresses the quantity of hours or mileage you shall test the functionality and can be affected also by the criticality of selected test routes.

2.7 Anomaly Detection

Anomaly detection (also called outlier or novelty detection in some contexts) aims to detect rare events that deviate significantly from the majority of the data or differ from an expected pattern. It is an active area of research, with increasing demand since it can be very useful in many different domains. Although classical ML methods (e.g., distance-based, ensemble-based, statistical algorithms) have been widely adopted in anomaly detection tasks, their performance is challenged when they are applied on IoT data streams [39]. In this context, some of the significant challenges occur due to scalability issues, high dimensional and heterogeneous feature spaces, feature interdependencies, cost of feature extraction, sparsity of anomalous events and imbalanced classes, and the difficulty to detect conditional anomalies (contextual, collective or time-dependent anomalies).

Deep learning methods have been very promising in learning useful representations from high-dimensional and heterogeneous data. They are scalable with big data volumes, adaptable in handling heterogeneity, and do not require cumbersome feature engineering, which enables end-to-end optimization of the whole task pipeline [39] [40] [41]. Moreover, representation learning of normality/abnormality is another advantage in deep learning methods, since

classical unsupervised methods can only estimate statistical deviations without obtaining any prior knowledge of expressive representations, which can be useful to generalize and detect novel anomalies [40]. Finally, deep sequential models are very effective in dealing with temporal complexity and capturing long-term dependencies, a desired property in IoT data applications in which time-series data are more closely related to collective or contextual anomalies than point anomalies [41] [42].

Deep anomaly detection aims to learn either feature representations or anomaly scores using neural networks. Extracted features can then be used with a downstream anomaly scoring algorithm in a disjoint learning setting. However, a more recent paradigm is learning feature representations of normality [40], in which a single model is used to both learn features and use the learned representation of normality to obtain the anomaly scores. A prominent choice following this paradigm is the autoencoder model, which learns low-dimensional feature representations and then its data reconstruction error is used to define an anomaly scoring function. The core assumption, in this case, is that the normal instances can be reconstructed more accurately than the anomalies. Autoencoders are straightforward in terms of implementation and training, while they are generic, allowing the integration of any deep learning network type in their architecture, e.g., CNN, RNN, etc., depending on the nature of the data (e.g., sequential, tabular, images). Some drawbacks might occur when the normal instances used in the training are not pure enough and contain anomalies. In this case, the model might learn a normal representation that is biased by several irregularities, resulting in weakness to detect such irregularities as deviations from the normal.

To adapt the autoencoder architecture to sequential data such as the IoT data streams captured in TEACHING, the implementation can use recurrent hidden layers similarly to the LSTM-AE architecture [43] [44]. The implementation of the LSTM-AE architecture comprises of two sub-networks, the encoder and the decoder, which are constructed with LSTM units since these models can track long-term dependencies in temporal or sequential data, such as sensor and time-series data. The computation flow is as follows: a multivariate input vector passes through the encoder which consists of one or more LSTM layers of progressively smaller dimensionality; the output encoding, is a compact representation of lower dimensionality, which is then fed into the decoder subnetwork, the final layer of which, is a reconstruction layer of the same size as the original input. The objective function uses the reconstruction error to penalize the difference between input and reconstruction.

After the network has been trained with normal instances, the reconstructions errors of the normal representation can be used to define the anomaly scoring function. The simplest solution is to define a hard threshold by taking for example the highest reconstruction error from the normal data and mark all new instances exceeding this error as anomalies. A more robust solution is proposed in [43] [44] the reconstruction errors **e** of normal data are modeled as a multivariate Gaussian distribution, and its parameters μ and Σ are estimated using Maximum Likelihood Estimation. The anomaly score for a new data point is then computed as $a = (e-\mu)^T \Sigma^{-1}(e-\mu)$, and a threshold over the likelihoods is learned by maximising the F_β. In general, continuous anomaly scores can be much more informative than just a binary label.

The anomaly detection learning module as part of the AIaaS is intended to be used by a cybersecurity application that will function as an intrusion detection system monitoring network traffic in TEACHING system. This functionality is aimed to enhance the dependability of the system against cyber attacks as described in D3.2 and D5.2. Nonetheless, the generic nature of the algorithm makes it suitable for other applications besides cyber threat detection within the

TEACHING framework. For example, anomaly detection can be applied to human biometrics data, or resources consumption measurements, runtimes, and other unlabeled time-series data.

3 AIaaS Architecture

This section focuses on the *AI as a Service* (AIaaS) software architecture that supports AI applications in TEACHING. Section 3.1 recaps the rationale of the approach, Section 3.2 lists the high-level requirements of the architecture, as they were refined after the release of D4.1. Section 3.3 provides an overview of the current architecture supporting AIaaS, while Section 3.4 discusses the data and metadata format adopted for communication within the architecture.

3.1 Rationale

The TEACHING approach to AIaaS on Edge and Cloud devices relies on designing reusable, portable AI application as a combination of composable, generic app-building blocks called Learning Modules (LM) and data sources. The rationale of the approach is that:

1. The LM building blocks can be separately ported to and optimized for different device HW/SW architectures, increasing their efficiency with respect to common metrics (performance, power consumption), allowing careful debugging and verification, as well as allowing to exploit specific features of the execution platform within the LM.

2. AI applications are more easily developed, reducing their overall complexity, increasing their reliability, and shortening the time-to-market.

3. AI applications effortlessly become as much portable as the LM supporting SW architecture is. That is, deploying apps on a plethora of Cloud ad Edge devices is allowed by making the focused effort of adapting the AIaaS support to those devices, without need of changing the apps and allowing different HW/SW devices to interoperate in a distributed software platform.

The aim of developing a dedicated support architecture for AIaaS in TEACHING thus requires choosing a trade-off between LM expressiveness and tailoring the LM to the HW. This is necessary in order to strike a manageable balance between achieving reusability of AI Apps across AIaaS implementations and easing the porting of the whole AIaaS architecture to new devices (this shall remain a mostly straightforward and manageable task except possibly for HW-specific optimizations). Two main abstract goals were held as reference "*lighthouses*" in the process of architecture design:

- *Allow adoption in TEACHING (being fit for the use cases):* The architecture must be portable and lightweight to suit the automotive and avionic use cases, allowing to build generic application with the suite of LM provided, while at the same time allowing to exploit specific hardware resources thanks to interchangeable implementations of the same LMs.

- *Allow reuse in different contexts:* The AIaaS supporting architecture shall be useful as a tool for porting AIaaS applications in different execution contexts, including the Cloud and various types of Edge devices (e.g., mobile units as well as fixed edge devices). Developing a full AI stack and development kit would be out of scope and would not get any adoption, thus the architecture needs to be designed exploiting existing, technologically relevant and/or industrial-standard AI frameworks at its core, namely:

- **TensorFlow (supported since the mock-up)**
- TensorFlow Lite
- WindFlow / FastFlow

The impact of the different choices with respect to the core AI framework and their implications on the AIaaS architecture design are discussed in the rest of this deliverable, specifically in this section and in Section 4.

3.2 High-level Requirements

We briefly summarize what the chosen AIaaS approach entails, stemming from its thorough description in D4.1 but also including further development after the release of said deliverable, i.e., results of the AI research activities and of the SW development and integration activities on the first prototype.

The TEACHING AIaaS architecture needs to support:

- LM execution (possibly on different supporting frameworks, according to the AIaaS implementations);

- the overall application deploy/manage/adapt/shutdown lifecycle and its impact on corresponding LM actions

- provide communications connecting its own local modules, that need exchanging:

  - structured data to be processed, or already processed; both batch and streaming communication modes are relevant for the AIaaS support;
  - metadata (as associated to the data);
  - model information (i.e., weights, coefficients that encode an AI model);
  - model metadata;
  - exception-like aperiodic messages (possibly with data payload) for specific uses within application (e.g., aperiodic out-of-band knowledge reporting, self-evaluation and issue detection and reaction);

- provide access to local storage, sensors, actuators, remote connections (exploiting the same communication mechanisms and modes already outlined): Remote connection is also a mean to implement distributed machine learning (DML). AIaaS currently **supports** this model but does not yet **implement** it. Each app in a specific AIaaS instance can be part of a larger DML scheme, but for the moment this scheme is not encoded in any specific LM, the architecture only provides the basic tools with respect to the task. A future research activity and a revision of the AIaaS and, most important, of the set of LM, may allow to systematize and simplify the creation of DML schemes among *multiple AIaaS App instances*.

- support a model where the reliability of (combinations of) LMs can be esteemed/evaluated via mechanisms that are built in the LMs and/or via additional, dedicated LMs, and can result in periodic/aperiodic actions and reactions within the app itself, as well as communication outside the specific app instance.

3.3 Overview

This section provides and overview of the software architecture supporting AIaaS in TEACHING. As it reports about an architecture already defined in D4.1, we aim at a self-contained presentation that does summarize key information from the previous deliverable while focusing on changes and advances with respect to that document. Specifically, we shall distinguish the overall design of the software architecture from its mock-up implementation. The mock-up was used to kickstart the prototype implementation and over time is gradually extended with new implementation of all the key modules.

3.3.1 Architecture Overview

In Figure 9 we show the current SW architecture of the AIaaS support, basically the same presented in deliverable D4.1, despite some minor evolutions. The overall design of the system remains the same: applications are provided in a mostly descriptive form (*Application Description*, heavily based on the composition of predefined *Learning Modules*) which is processed by the *Application Translator* component. The translations employ available implementations of the Learning Modules stored locally in the *LM Library* to produce executable code for the *AI framework* as well as initialization data, parameter/hyperparameters for the computation, and support information for the *Application Runtime* to steer the application execution. The computation mostly happens within the chosen AI framework, with data flowing through internally[3] to the AI framework as well as outside of it. The data is routed internally to the AIaaS architecture by the *Data Brokering* component, which also interfaces:

- to the *Sensors API* group of devices, to receive data from the physical part of the CPS;
- to the *Local Storage API* group of components, in order to save and retrieve data, and
- to the *External communication Interface*, in order to allow data exchange with remote systems;
- finally, the AIaaS support and the applications it hosts can influence the physical part of the CPS via the *DMU* module[4].

[3] Note that the *AI data bus* is an abstraction, an Event Bus design pattern within the AI Framework that may not correspond to a software component in the final implementation.

[4] Within TEACHING use cases, the influence on the physical world is limited to setting parameters of the dependable systems that actually interact with the physical world, e.g., setting the driving mode. This can affect the status of the driver and passengers, but is a safe change to apply at any moment. Nevertheless, the architectural design takes into account that the DMU interacts with dependable system and may thus refuse any demanded action.

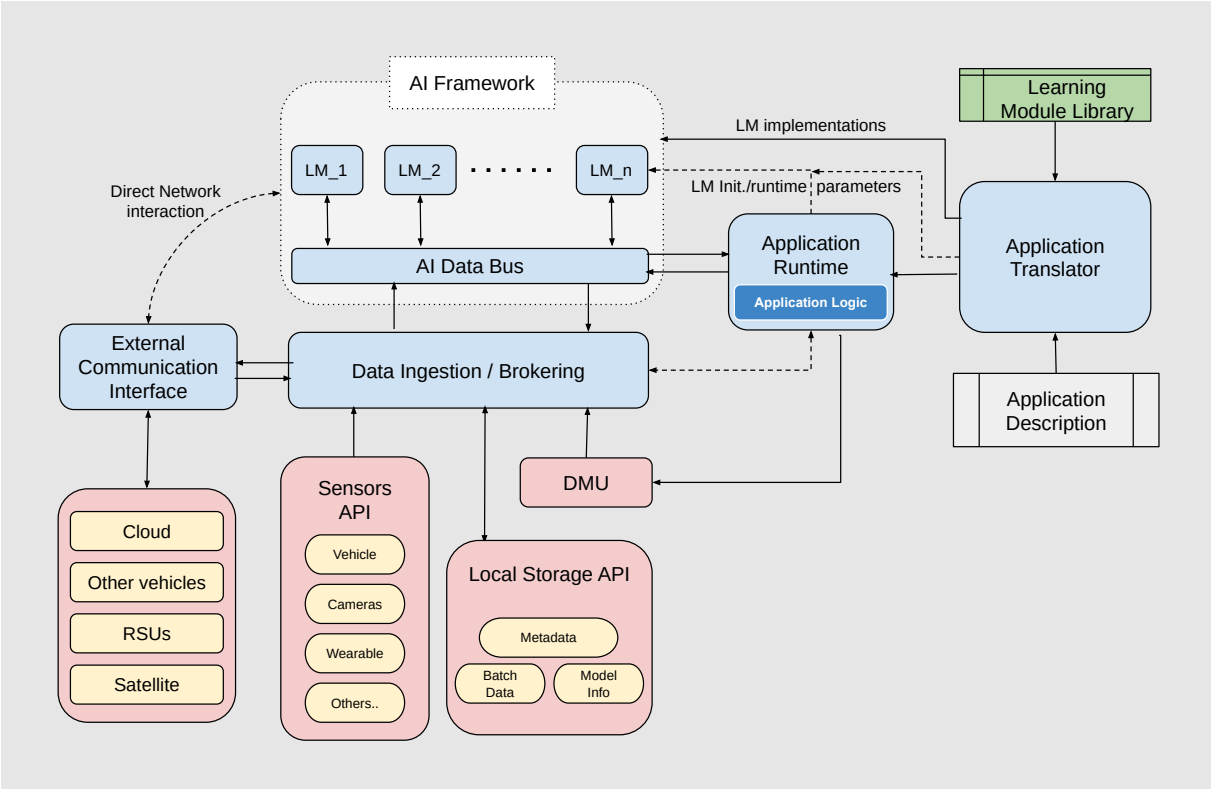


Figure 9 AIaaS SW Architecture Diagram, current design

With respect to the design presented in D4.1, a direct interaction between the code in the AI framework and the External communication interface is being studied (see Section 4.5).
The Figure 9 shows what the final design of the AIaaS system is expected to be. In the rest of this Section 3 and in Section 4 we discuss the features of the AIaaS support, as well as the design and implementation of its components. It is thus necessary to also show the current implementation status of the whole architecture, which we present next in Section 3.3.2.

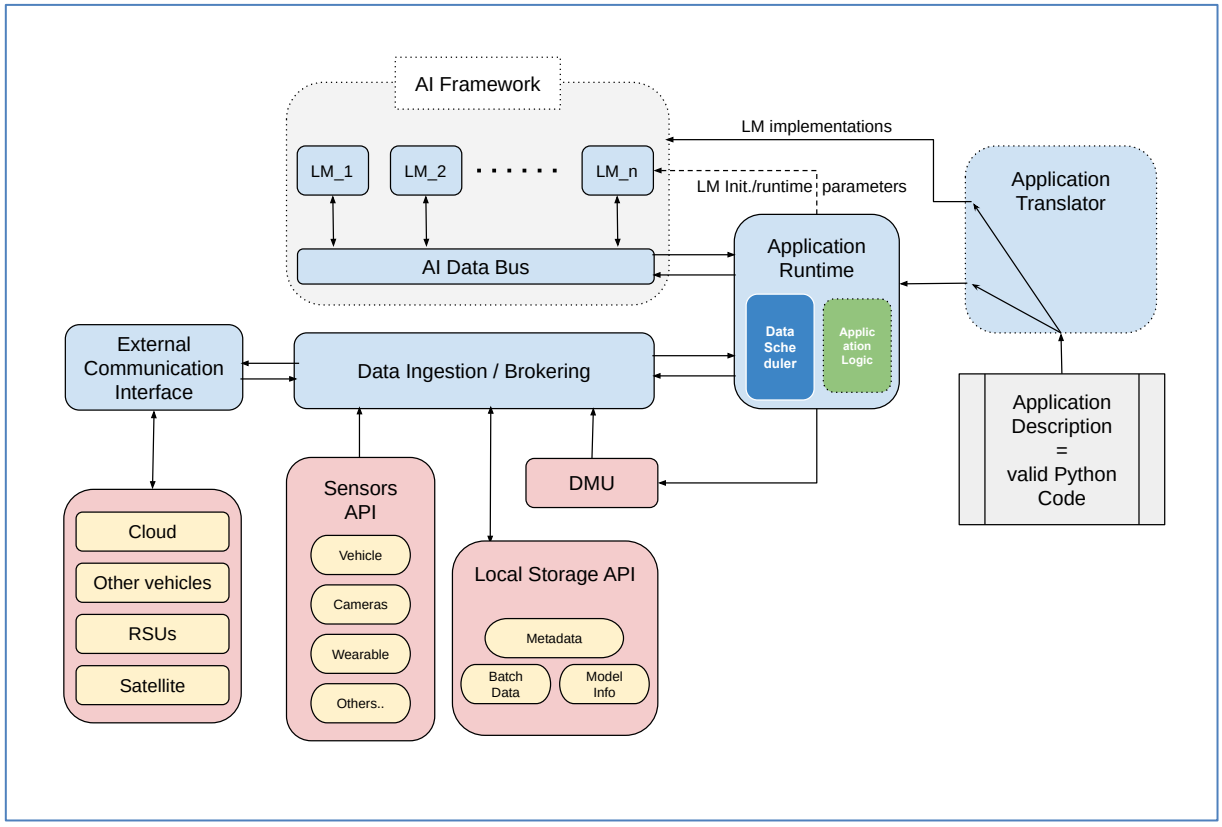


Figure 10 AIaaS SW Architecture Diagram, M18 Prototype

3.3.2 Prototype Architecture

With respect to the current version of the architecture, that we presented in Figure 9, the mock-up has simplifying assumptions. The M18 prototype (see Figure 10) that evolved from the mock-up follows those assumptions. A few modules and interfaces are not yet needed, some have not yet been developed or integrated. The prototype design differs from the reference one in the following aspects:

1. As we are experimenting with the *Application Runtime* and *Application Logic* code within the prototype apps (that are for this reason fully valid Python code) the following modules are different or are not yet implemented (see Sections 4.2 and 4.3):

- Information concerning the application description is embedded within the application itself via self-describing objects that are programmatically built. The app is thus fully valid Python code, not a custom description (an alternative way of seeing this is that the application description is part of the Python code of the application).
- Application Translator is not yet used.
- The LM module library is not yet used.
- Application Logic is still embedded within the Application Runtime Module.
- The application runtime code is also embedded within the application.

2. There is no connection between the Data Ingestion/Brokering and the AI Data Bus, data exchanges with MQTT are mediated by the Application Runtime's scheduler (see Section 4.2).

3. There is no direct connection between the AI framework and the External Communication Interface (see Section 4.5).

3.3.3 Application Description

The overall purpose, requirements, and rationale of having a formal, declarative document format detailing the structure, constraints and intended behaviour of an AI application have been discussed at length within deliverable D4.1 Section 4.2 (specifically sub-Section 4.2.2), which we refer to. The content of the application description can be summarized as:

1. A list of the LMs that compose the application,
2. LM parameters and hyper parameters
3. LM initialization data / models
4. A description of all communications required among the LMs, with sensors, data storage and outside the AIaaS instance (the *Application Graph* of typed data streams)
5. constraints and preferences for the LM execution, to help chose the best LM implementation
6. code snippets, if needed, that deal with special cases and exceptions specific to the application semantics in an unstructured manner. They are to be executed as part of the Application Logic.

While the abstract design of the Application Description has not changed, the first implementation in the M18 prototype relies on the information being provided by Python code. The fully descriptive format is currently represented by programmatically built Python objects that specify the necessary details. For the sake of clarity, we summarize some implementation details that are described later on, in Section 4:

- Each LM is an instance of a Python object and is provided parameters at creation.
- The Application Graph, i.e., the set of communications streams among the set of LMs, sensors, storage elements et cetera, is also a Python object programmatically created when initializing the actual computation, before the application can start.
- Code snippets that implement custom function for the application are directly provided as Python functions within the application code (see Section 4.2).
- Once the set of objects describing the whole of the application structure has been created, the application can be started.

As such, there is not yet a strong separation between the declarative description of the app and its imperative code.

3.3.4 Data Routing Definition in the M18 Prototype

At a high level, an application developer can declare an application that uses the TEACHING framework by providing a few instructions in a Python file. We refer to the example application *stress.py* that performs stress recognition and can be found within the main AIaaS software repository[5].

[5] https://teaching-gitlab.di.unipi.it/v.lomonaco/ai-toolkit/-/blob/master/applications/stress/stress.py

In order to route data, the application developer declares the sensors and learning modules that needs to be used, and then defines the *routing* of the application, in the form of a list of edges forming a *computational graph* of information flow. For example, the code

```
app.route([
    (eda, teaching.output, {}),
    (eda, lm, {}),
    (lm, teaching.output, {})  # Exit point
])
```

instructs the runtime to route data from the *eda* sensor to the output (for display) and to a learning module named *lm* (for prediction). The data emitted by *lm* is then further routed to the output. The above code is a declaration for setting up the environment, it specifies some links within the Application Graph, but does not cause yet any data to flow.

When the application is run, the outputs (in our example, the *eda* data and the predictions from the *lm*) can be obtained by the last LM in the chain as soon as they are available.

```
while True:
        first_output, second_output = await app.output()
```

Data from different sources may be available at different times, and a generic LM may need all of its data sources (or a specific subset of them) before it can start a computation. The abstract definition of data routing provided by the Application Graph is thus connected with the execution constraints of each LM within the application.

The pipeline graph outlined above is a very simple example where the data availability constraints are obvious. Different synchronous and asynchronous computations among the set of LMs in more complex Application Graphs are possible at application runtime. They can be managed according to the dataflow[6] computing approach. The actual execution order (the firing order, in dataflow lingo) is chosen by the Application Runtime component (see Section 4.2).

3.4 Data and Metadata Formats

A common data format is desired for data that can flow through the LMs and AIaaS architecture components. Requirements on the Data format were clear since previous Deliverable D4.1. The format had to provide a common data representation that is both architecture-agnostic and language-agnostic enough, and that causes a low data conversion overhead, in terms of:

1. absolute performance and memory occupation – especially on constrained HW
2. performance on the critical pat – no data conversion should be required in the most demanding part of the AIaaS architecture, namely the AI framework
3. added complexity of SW – complexity would impact on code maintainability and system dependability

As transferred data need to be associated with a type system and can be structured (e.g., for passing tensors with data and model weights), from point 3 it follows that serialization, deserialization and data conversions have to be performed via support libraries / off-the-shelf

[6] https://cloud.google.com/dataflow

code, to avoid the pitfall of rewriting common code. Finally, in the same format it should be possible to store, along with the data, suitable metadata to support AI applications.

In the following Sections 3.4.1 and 3.4.2 we outline the two solutions devised for data that move across the whole AIaaS support, and data that is local to the AI framework.

3.4.1 Data format and message structure outside the AI framework

Outside of the AI subsystem (e.g., TensorFlow) and specifically when using the Data brokering system, we plan to use Protocol Buffer as the main data format.
Protocol Buffers are the native format of TensorFlow and are publicly documented[7] by Google. They match the stated requirements of standardization, limited overhead, parsing and conversion code being available off the shelf for multiple programming languages.

As data may need to be routed to different LMs, to be retrieved at a later time and upon specific conditions or by a different app than the originating one, metadata associated to the data is needed. The foreseen metadata needs include:

3 application/LM that created the data

4 time of creation

5 data type and structure, if not encoded in the data format

6 source / destination component and module for data to be routed

To support attaching metadata to the data, we plan to exploit ProtocolBuffer *custom options***,** natively supported by Protocol Buffer since v2.

3.4.2 Internal data Format for the AI framework / AI data bus

For the sake of communicating data among LMs, a simpler data format is used in the M18 prototype, the *DataPacket* wrapper class. Within the AI framework, data flowing through learning modules still needs to be annotated to support the functionalities offered by the framework. For example, most of the times the data should have an associated timestamp, which is used to synchronize different streams within the framework. Another annotation indicates semantically what kind of data is flowing through an edge, for example “normal” data or “label” information for training.

As of now, the DataPacket wrapper includes annotations for the timestamp and for the type. The DataPacket:

- is a Python structured type,
- it provides storage for array data with labels, associated timestamps and type information
- it does not enforce a serialization at the LM boundaries for TensorFlow and TensorFlow lite
- it can be easily used by the *AI bus* implementation (the AI bus is currently the function acting as data dispatcher within the Application Runtime).

As the M18 AIaaS prototype is fully Python based, the Datapacket can be used also outside of the AI Framework instead of the designed ProtocolBuffer format. Whether the DataPacket is

[7] The ProtocolBuffer interfaces are documented here: https://developers.google.com/protocol-buffers, and the custom options extension docs can be found at https://developers.google.com/protocol-buffers/docs/proto#customoptions

still going to be used outside the AI framework, and in what implementations of the AIaaS besides those based on TensorFlow, it is a matter to be discussed after the M18 prototype finalization activities.

4 AIaaS Platform Components

This section describes the current status of implementation and choices of all components of the AIaaS system, as they are in current M18 prototype.

For the sake of clarity, we underline again the distinction between LMs, which are the building blocks of AI applications, and the software components that compose the AIaaS architecture *supporting* the execution of applications made from LMs. While from a SW engineering viewpoint both kind of entities are software components, when we speak of components in this deliverable, ad definitely in this section, we are referring to the software components that compose the AIaaS support architecture.

The AIaaS architecture must support LM modules written using **TensorFlow**, **TensorFlow lite**, and FastFlow skeletons. Current AIaaS design can only support one such framework at a time (i.e., no mixing of frameworks when combining LMs into an application).

The main programming language we support inside AIaaS application (i.e. as a glue language in addition to the LM blocks) is Python. We foresee that this choice will not need to be revised even when supporting different languages and execution modes (compilation vs interpretation) for the implementation of the LMs. We assume Python execution is always available on all project HW/SW platforms. Snippets of applications can be encoded in Python, if the platform does not support a full python stack, Python-to-C translation with Cython can be considered.

The rest of this section discusses the AIaaS components currently existing in M18 prototype and depicted in Figure 10 (Section 3, on page 32). Section 4.1 starts from the AI Framework that the AIaaS leverages for LM execution. Section 4.2 describes the Application Runtime and Application Logic, it discusses the implementation of the Runtime as well as of the data flow and computation scheduling within the AIaaS architecture. Section 4.3 discusses the designed function of the Application Translator (although its current implementation is still minimal).
Section 4.4 (Data Ingestion/Brokering) and Section 4.5 (External Communication) focus on the design and the current implementation of the components providing communication support respectively within the AIaaS system and with systems reachable via the outside networks.
The last three sections describe the components providing access to the physical part of the CPS the AIaaS subsystem lives within, namely the set of its sensors (Sensors API, Section 4.6), the local permanent storage (Local Storage API, Section 4.7), and the Decision Management Unit, the component allowing AIaaS apps to “act” on the physical world (Section 4.8).

4.1 AI Framework

The **AI framework** hosts the learning modules as they are instantiated, linked together and executed to perform any Machine Learning tasks needed by the AI applications. This module leverages existing, in most cases Industry-standard ML technologies in order to provide the implementation of the LM functionalities.
The choice of the ML technology used to implement the AI framework is not cast in stone: it is a parameter of the AI Framework implementation. All LM modules are implemented on top of the ML framework contained inside this module. LMs can have different implementations (with the same semantics) over different ML frameworks. According to the actual choice made in the AI framework implementation, the corresponding set of LMs is to be used.

The internal implementation of the AI framework is surrounded by a thin interface layer whose purpose is to provide the interface of the AI Framework unchanged despite the different choices made for the internal implementation (TensorFow, TensorFlow lite and so on). The interface layer, where needed, is made up of a set of adapters translating different data representations and possibly different internal APIs as needed. The same set of high-level LM semantics and interfaces are expected to be provided independently of the inner AI framework. At the moment, given the limited nature of the current AIaaS system prototype supporting only Tensorflow, this functionality is not implemented and the AI Framework constitutes mostly an abstract entity.

4.2 Application Runtime

The Application Runtime implements all tasks that are common support of LMs and are common to all TEACHING apps.

As stated in D4.1, the **Application Runtime** component manages the main execution workflow of an AIaaS application. Its core functions include:

1. the instantiation of the underlying **AI framework** that hosts the learning modules;
2. instantiation and execution of the LMs of the application;
3. configuration of the AI data bus to correctly route the data stream to the relevant LMs.
4. manage the Application Logic, triggering and providing data to its function.

This implies several key interoperation features that depend on the assumptions made on a specific implementation of the AIaaS support. We describe the design of the AR and then discuss its current implementation assumptions and simplifications. Key differences set apart interpreted, Python-based frameworks (i.e., TensorFlow and TensorFlow lite) from those based on offline compiled code, like FastFlow or WindFlow are (they are both based on C++ compilation).

Instantiation of the AI framework – the initialization and allocation of resources for the AI framework is influenced by the specific AI framework employed. Python code (Tensorflow, Tensorflow-Lite) may rely on different HW and SW prerequisites being available, which must be checked at initialization time. For compile-based frameworks the initialization shall check that compiled version of all the LMs are available in executable form.

LM instantiation and execution – the AR needs to support LMs for:

- **Instantiation** – Setting up interpreted Python code for execution poses a lesser problem, while executing assembling already-compiled code into applications requires the runtime to either manage binary linking or dynamically loaded libraries of LMs. The execution of LMs.
- **Execution** – computation in the LMs requires data. The runtime is in charge of bringing data to the LMs and firing their execution, exploiting the routing information provided by the Application Graph (currently implemented as a Python object). It shall be noted that LMs have a dataflow-like semantics, where execution is possible when all the needed data inputs are present. Dealing with real world data without risking stalls requires a specific semantics defining which inputs are required, and how to deal with missing optional inputs (e.g., keep previous data, send a predefined empty data). Two extreme approaches are possible, the more general and concurrent one is to provide each LM with the information needed to directly interact with the Data Brokering,

implementing most communications and all nondeterministic input handling directly within the LMs. In the diametrically opposite approach, the one that the AIaaS M18 prototype adopts, all Data Brokering interactions are instead kept within the Application Runtime, which forwards the actual data to the LMs and implements the semantics of non-deterministic control.

- **(Hyper-)Parameter setup/update** – several LMs sport a very different behaviours according to their parameters. Choosing, forwarding, and possibly updating LM parameter is a task that depends on AIaaS and LM implementation, but whose semantics are deeply tied with the application ones.

AI data bus configuration – the AI data bus is intended as a low overhead mechanism for LM-to-LM communication, skipping most or all needs of data conversion (as the LMs within a single AIaaS are implemented on the same AI framework) and skipping unnecessary synchronizations when the LMs cooperate in a straightforward way (e.g., simple pipelines). Setting up this kind of communication channels is done by the AR at LM initialization.

Application Logic – citing from deliverable D4.1, "*the Application Logic is a per-application software module. It embodies those parts of the application which are in charge of handling all special cases and actions that depart from the main workflow of the Application Runtime (deployment, data collection, and data analysis)*". Due to the currently experimental status of the Application Runtime as well as of the test applications, where each prototype is still including all boilerplate code from the AIaaS, the Application Logic module is not yet defined within the Application Runtime, whose initial design is still valid but will be implemented later on, after the Application Translator and Application Runtime components are more mature.

4.2.1 Application Runtime Implementation

The current design, implemented in Python and stemming from the initial mock-up, is a simplified version of the Application Runtime exploiting the initial assumption of Python-only code and skipping modules which are yet in development, or are not useful for the first prototype.
Specifically, the current implementation of the AR does not yet host a formally defined code section to implement Application Logic. As stated before that specific sub-module is not currently defined:

- Data messages from the various sources[8] are pushed as data packets to the LM by the AR, that is also in charge of getting back the results, both for ordinary data streams and aperiodic (exception-like) ones.

- This also entails that the dataflow firing policy of the LMs is currently implemented within the Application Runtime, and all interactions of the LMs with the Data Brokering are mediated by the AR.

- The application start-up and initialization is defined within the application itself as Python code as boiler plate (still evolving) code. As the features of the AR evolve and settle down, the initialization code will move to the AR. The Application Translator

[8]As already described in D4.1 and detailed in this document in Sections **Error! Reference source not found.** and 4.5, sources are actually identified by topics in the publish/subscribe networks providing local and global connectivity.

component will then parse application declarations and hook them into the appropriate calls to the AR.

4.2.2 Data Flow and Activity Scheduling in the M18 Prototype

The Application Runtime moves the data from the sources (typically sensors) via the Data Broker to the output(s) by following the routing instructions provided by the application developer, and taking care that each module that has all incoming edges with data has in turn its execution method called, e.g., node.train(x,y). This logic is currently implemented in the function "tick" of the data_bus module according to the following pseudocode, which is endlessly executed until the application stops/is stopped.

```
def tick():
    for each node in the routing graph:
        if the node has no incoming edges:
            # it is a source such as a sensor
            buffer[node] = node.read()  # Returns a buffer of
      readings
        else if all nodes (x) in the incoming edges (x)->(node) have
                data in their buffer:
            merge and sync the incoming data
            if node == teaching.output:
                return the data
            else:
                call the node with the merged data
        else:
            process this node later, when all inputs are available
```

4.3 Application Translator

The Application Translator software component is designed to receive a description of the application (an Application Description document, see Section 3.3.3 and D4.1 Section 4.2.2) that is mostly of declarative nature and relies implicitly on the set of LMs to be available in the current AIaaS instance. The Application Translator then plays two roles:

1. it generates and configures the ready-to-be-executed instances of the LMs that are part of the application, processing the declared application structure. This entails in turn:

    - extracting the information about what LMs are needed by the app;

    - identifying suitable LM implementations that are made available in the LM library. When choosing the LM implementations the AT will consider both hints provided by the application, and the features provided by the LM hosted by the library, aiming at the best match according to the application-specified metrics (e.g., absolute performance, power usage reduction, self-evaluated reliability and so on);

    - composing them in the way needed by the AI framework that is in use. Depending on the use of Tensorflow, Tensorflow-Lite, WindFlow or other AI frameworks, the LM composition can be just a matter of trivial textual

juxtaposition of source code, it may require class instantiation and template-based programming, or it may need runtime linking and dynamic library provisioning).

2. it provides the application runtime component with the Application Logic elements as derived from the Application Description. As the standard part of the Application Runtime component (see previous section) is progressively extended and refined, exceptions and special cases will emerge that AI applications need to manage in a code-efficient way. The Application Translator will extract such snippets of code from the declarative form of the application and pass them to the Application Logic sub-module of the Application Runtime.

At the current stage of development, the M18 prototype is Python only and all applications are still fully valid Python code, embedding all the AIaaS support code in their own classes. As such, no translation is needed, and no special logic is extracted or generated. The current Application Translator is a no-op class and both the Application Logic and the LM library not yet in use within the prototype. In the next months these modules will be added, with a progressive implementation of the two roles outlined in this section for the Application Translator.

4.4 Data Ingestion / Brokering

As described in previous deliverable D4.1, all the components of the AIaaS architecture locally communicate with each other via the Data Ingestion/Brokering component. This component is built around the MQTT protocol, based on the Paho libraries[9] available for both Python and Java. Among the reasons for choosing MQTT are that it is a lightweight pub/sub protocol, well supported by, and used on, embedded and mobile devices, as well as by conventional OSes (and thus on Clouds). MQTT is thus a reasonable middle ground and a flexible gateway toward the more powerful, expressive and extendable Kafka protocol that is employed in the overall communication architecture, as we describe in this document and in deliverable D2.2. Figure 11, which we include from D2.2 for the sake of readability, shows the overall organization of the TEACHING pub-sub network.

The AIaaS components (as well as, possibly indirectly, any learning modules instantiated onboard the mobile node) can exploit MQTT both to communicate with each other and to access the whole infrastructure. The modules are thus provided with mechanisms that via topic-based addressing allow several forms of communication, including as elementary cases point to point, broadcast, multicast, and gather communications. To actually send/receive data, that will be transmitted through the broker, the AIaaS support components use various forms of the subscribe and publish functions in the library.

4.4.1 Data Brokering Implementation

We describe the Python interface to the Data Brokering. The **subscribe** method accepts 2 parameters: a topic or topics and a QOS (quality of Service, with values within 0-2) as shown below.

```
subscribe(topic, qos)
```

1. Method 1- Uses a single topic string. This is an example function call.

[9] https://www.eclipse.org/paho

```
client1.subscribe("house/bulb1",1)
```

2. Method 2- Uses single tuple for topic and specifies QOS level

```
client1.subscribe(("house/bulb2",2))
```

3. Method 3- Used for subscribing to multiple topics, using a list of tuples

```
[(topic1,qos),(topic2,qos),(topic3,qos)]
client1.subscribe([("house/bulb3",2),("house/bulb4",1),("h
     ouse/bulb5",0)])
```

The subscribe function returns a tuple to indicate success, as well as a message id which is used as a tracking code.

```
(result, mid)
```

The MQTT broker/server will acknowledge subscriptions, which will then generate an `on_subscribe` callback. The prototype of the callback function is shown below.

```
on_subscribe(client, userdata, mid, granted_qos)
```

The **publish** method accepts 4 parameters. The parameters are shown below with their default values.

```
publish(topic, payload, qos, retain)
```

The payload is the message you want to publish, the topic is where you want to publish the QoS is the reliability for the sent messages, the flag retain "on" allows the broker to store the last message and the corresponding QoS for that topic.

```
client.publish("house/light","ON","1","off")
```

Before a client can start publishing or subscribing a simple **initialization** is required: creating an instance, connecting to the broker, and finally publishing/subscribing to data, as in the following example:

```
broker_address="192.168.1.184"
client = mqtt.Client("P1") #create new instance
client.connect(broker_address) #connect to broker
client.publish("house/bulbs/bulb1","OFF")
client.subscribe("house/bulbs/bulb1")
```

4.5 External Communication Interface

The external Communication Interface component allows local AIaaS modules to connect to the network and communicate LM-generated/required data with Cloud services and other instances of the AIaaS architecture. This module exploits a custom Kafka client which is developed in the context of WP2 and is used as a bridge between the local MQTT broker (within the Data Brokering component) and the global TEACHING pub/sub network. The implementation of the Kafka client that we use in the mock-up and in the first implementation of the AIaaS subsystem is coded in Java.

The bridge is configured with two lists of topics that are to be forwarded in both directions, defining what kind of information is allowed to flow and thus providing isolation from the

network to the local environment (the applications and sensor data within the AIaaS) as a basic mechanism for keeping any kind of sensitive and private data local to a specific CPS.

The pub/sub mechanism provided by the External Communication Interface is supported within WP2 at the platform scale, as it is shown in Figure 11 (which is also in D2.2). The left part of the figure shows the (simplified) edge-based AIaaS architecture and its interaction with the Kafka-based network. The centre and right portion of the figure show the deployment on top of Near-Edge and Clouds of the networking and computation support.

The basic mechanism already provides the means to coordinate multiple instances of the AIaaS support, either on Cloud or Edge and Mobile devices. It thus allows to implement federated and distributed learning algorithms as a set of application deployed on multiple AIaaS instances. The approach is exploited in the first release of WP4 demonstrators and described later in this deliverable. Additional features are being studied and developed, which complement the fundamental approach:

1. Providing the Kafka-MQTT bridge client with the ability to exploit specific message fields to programmatically generate sub-topics on the receiving side, thus allowing for sender-controlled message routing on the receivers CPS. This specific set of features can help structure complex applications and implement distributed/federated learning patterns.

2. Allowing LMs to directly interact via Kafka (e.g., skipping the Data Brokering and MQTT). The approach is being studied as it represents a different trade-off that may be interesting in the context of “core” instances of the AIaaS, that is deployed as services on top of full-blown Cloud resources. Figure 11 on the right shows two examples of LMs, one interacting via the bridge client, and another one directly via Kafka. Adding such an interface can make more complex the model and implementations of LMs but may allow to exploit Kafka-specific APIs to provide the LMs with advanced support for distributed/federated learning.

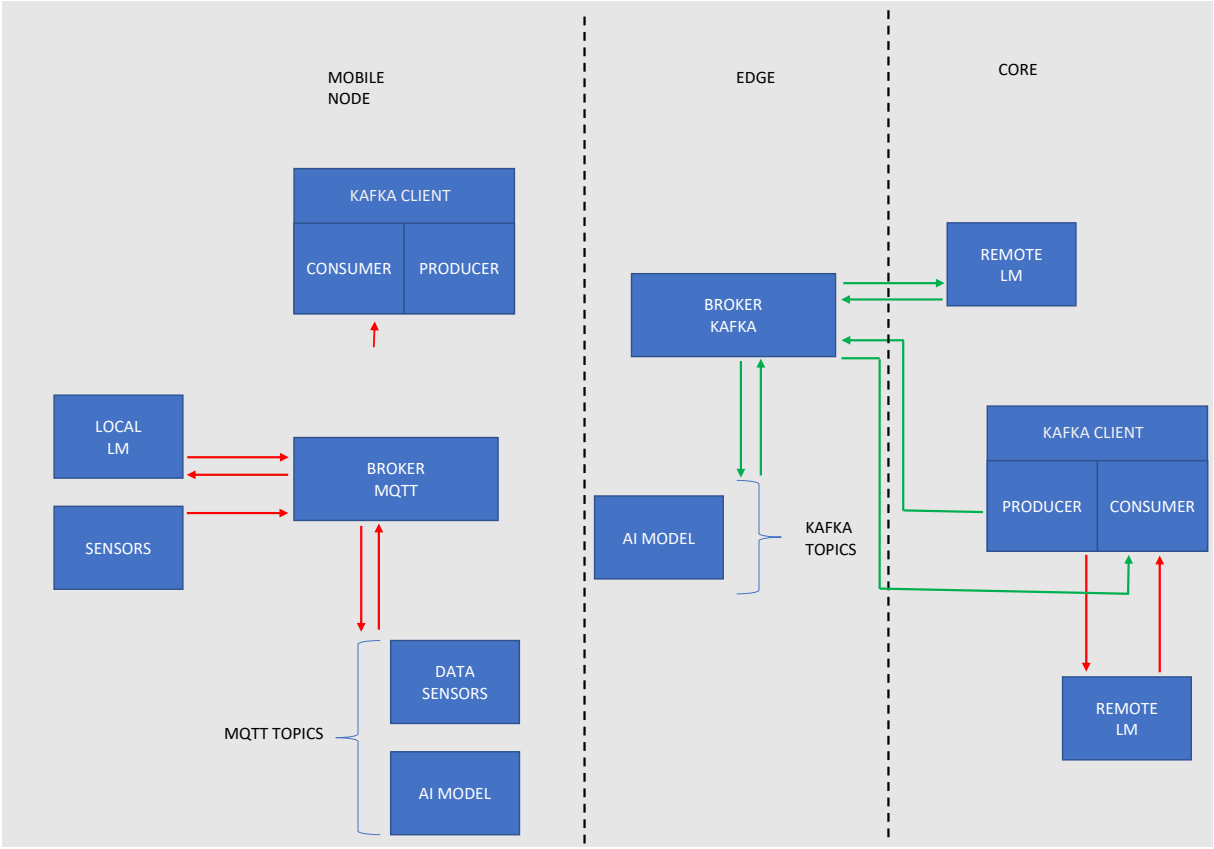


Figure 11 Overall PUB/SUB Communication Organization spanning WP2 and WP4 (from D2.2)

4.5.1 External Communication Interface Implementation

We only provide a short description of the implementation for the sake of completeness. For further technical details we refer the reader to deliverable D2.2, where the full description of the bridge pub/sub client is provided.

The client is implemented in Java, and in current prototype is exploited via Java and Python. It includes two connectors relaying messages from MQTT to the Kafka network and vice versa. The Kafka Connect **source connector** reads data from MQTT and publishes them to Kafka, while a Kafka Connect **sink connector** reads data from Kafka and publishes them to MQTT. Both connectors act on a configurable list of pub/sub topic trees, that works as an allow list for the data transfer by controlling the subscription operations of the bridge. The allow lists can match either single topics or whole subtrees starting with a given topic.

The external communication interface was tested and verified during the integration activities but is not always used in the experiments presented in this deliverable. Some tests rely on a simpler interconnection of multiple AIaaS instances obtained by sharing the MQTT broker. The MQTT architecture does not allow this kind of approach to scale to fully distributed, geographically dispersed settings, but the choice eased separate development and debugging of the AIaaS and LMs features.

4.6 Sensors API

The role of the Sensors API is to wrap the actual sensors that publish their streams to a message broker in a callable API that can serve data upon request from whichever module asks for the

respective data. In order to implement this, the sensor API maintains a data buffer, which collects data as they arrive at the message broker. The buffer keeps the latest messages from each sensor. When a request to read data from the Sensor API is made the API returns the respective set of latest messages and automatically empties the respective queue in order to receive new messages from this sensor. This guarantees that the Sensor API provides the latest readings for a sensor every time it is called.

Every time a new sensor is added to the TEACHING platform an instance of the SensorAPI is instantiated in order to provide access to the sensor. The constructor (**init**) also defines the size of the buffer. The **open** method creates an instantiation of a connection to the message broker to the respective topic of each sensor.

The **read** method returns the content of the buffer and clears the buffer. Figure 12 shows the basic operations scheme of the Sensors API in the AIaaS system. In order to provide room for scalability, we assume that the drivers needed for each case are available in order for each sensor to be able to publish on the MQTT bus, so these drivers can not be part of the AI-toolkit. In our case, all the appropriate drivers have been developed to support our use cases and demo applications.

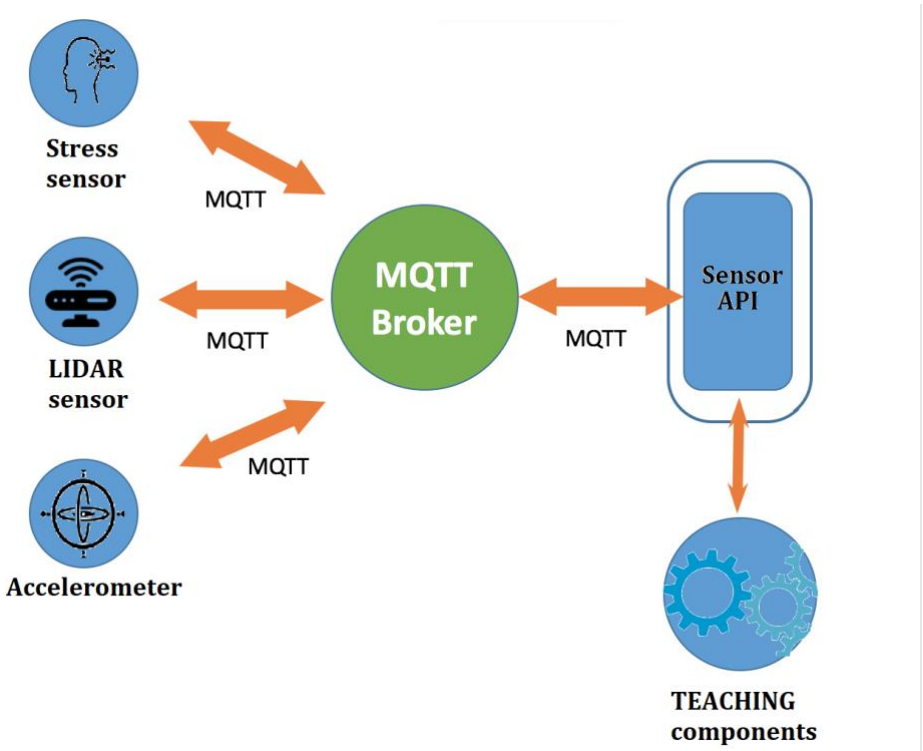


Figure 12 Sensors API in the AIaaS system.

4.7 Local Storage API

The aim of the Local Storage API is to provide the other TEACHING components with stored versions of the ML and AI models and also allow the long-term storage and reuse of newly

trained models. Apart from models, the Storage API allows to store configurations, temporary files and caches, to store results and any other custom binary object.

For this purpose, the Local Storage API maintains an SQLite (SQLiteStorage class) for storing the various objects (instances of an Item class), which have an id, a name and description, a storage type, a timestamp and the filename where the actual object is stored. The API provides methods for adding items to the storage, retrieving them from the storage, getting their metadata by id or name and also for removing items from the storage. Finally, it has methods that allow to store the item to the disk or retrieve it from the disk, but this functionality has not been used in the current mockup.

4.8 Decision Management Unit

The aim of the DMU is to provide an interface for communicating with the action units of the vehicle.

The only interfaced action unit so far is the driving profile selector, so the DMU provides methods for connecting the DMU with the vehicle (constructor), setting a profile change (method set_action), an asynchronous listener for consuming new signals that arrive from ML modules, and a publish method for sharing the action with other modules. All the communication with the DMU is directed through the message broker.
Currently, for the purposes of the mockup, we have developed a set of functions that have been used in the demo app that sends the LM outputs to the message broker and that may implement the application logic as well as the drivers needed on the autonomous vehicle side to be able to receive the DMU commands and apply these commands to the vehicle itself.

5 AIaaS Learning Modules

In this section we provide a detailed list of planned LMs APIs. A Learning Module (LM) is just a set of utilities to satisfy a specific learning goal. This means there is not pre-defined abstraction level for all the modules which can provide out-the-box functionalities for classic learning algorithms (e.g., RNN), specific tasks (e.g., Classification), learning paradigms (e.g., Continual Learning) as well as orthogonal learning features (e.g., Privacy-Preserving Learning).

Learning modules can be divided in two main categories: 1) *Standard LMs* and 2) *Support LMs*. The first type of learning modules are objects that can be instantiated in a computational graph defining the application logic; the second are objects that can be instantiated independently from the application logic and offer specific functionalities that are often used by Standard LMs to address a specific need. In Table 6 the planned LMs to be offered in the AIaaS system are reported.

Table 6 Table of Learning Modules available in the AIaaS

Name of the LM(s)	Description	Standard / Support LM
Time series RNN	Learning module implementing sequence learning and classification capabilities in teaching and based on standard Recurrent Neural Networks (RNN)	*Standard*
Time series RC-ESN	Learning module implementing sequence learning and classification capabilities in teaching and based on standard Reservoir Computing techniques, in particular Echo State Networks (ESN)	*Standard*
Federated Learning	Learning module for implementing federated learning applications: based on how it is instantiated it can also work as the centralized server in charge of the synchronization.	*Standard*
Continual Learning	This module offers basic functionalities to the other LMs to handle ever-changing data distributions and update a prediction model efficiently.	*Support*
Reinforcement Learning	The Reinforcement Learning module offers the main	*Standard*

	functionalities for developing applications learning from sparse rewards instead of direct supervision.	
Privacy-Preserving Algorithms	The privacy preserving learning module offer support functionalities to augment the privacy of the developed applications and estimate quantitatively the eventually leaked privacy	*Support*
Dependable AI	The dependable AI LM offers basic functionalities to ensure the dependability of the developed applications	*Support*
Hyper-parameters Selection	Hyper-parameters selection utilities that ca be used black-box functionalities by standard LMs.	*Support*
Anomaly Detection	Basic support functionalities for anomaly and cyber-attacks detection	*Support*

The rest of section is organized as follows: Time Series RNN and RC-RNN are detailed in Section 5.1 and 5.2, respectively. Section 5.3 describes more in detail the Federated Learning LM while Section 5.4, Section 5.5 and Section 5.6 introduce three different support modules namely the Continual Learning, Privacy-Preserving and Dependable AI ones. Section 5.7 details the Hyper-parameters Section support LM and Section 5.8 the Anomaly Detection one. Finally, Section 5.9 present the current state of the design for the Reinforcement Learning LM.

For each Support or Standard LM the currently planned (or partially implemented) API is described in more details. *These APIs are not cast in stone: while they provide an important first step supporting the mock-up AIaaS design and implementation, they may be subject to change and further refinement with the progressive development of the AIaaS functionalities, towards the preparation of deliverable D4.3.*

5.1 Time-series RNN

This module provides a basic interface for deploying *Recurrent Neural Networks* into any TEACHING application with a straightforward and automated interface. The main rational for Standard LM is to be as agnostic as possible with respect to the *Application Graph* they are part of. This approach favours modularity, reusability and easy-of-use. A RNN can be easily instantiated to handle incoming time-series both for training and inference. The high-level API provided in this LM allows even less-experienced software developers to tackle complex AI tasks with ease.

5.1.1 Execution modes

This LM has two execution modes:

- *Training*: modality to train the underlying prediction model.
- *Eval*: modality to use the prediction model in inference only.

5.1.2 Input and output

The LM expects the data to be formatted as Numpy[10] tensor data of the following format (*batch_size*, *input_features_size*, *target_size*). If *target_size* equals None, the problem will be assumed as a **sequence learning** problem, a **sequence classification** problem otherwise. The LM outputs the success or error states in the training mode (after each call), while it returns also the predicted tensors (*batch_size*, *target_size*) for the eval mode.

5.1.3 List of API calls

Here we list the main API of the LM:

- ***init****(n_layers, n_neurons, optimizer_hyperparams) -> State*
- ***train****(input_tensor) -> State*
- ***eval****(input_tensor) -> predicted tensor*

5.1.4 Implementations of the LM

The implementation of this module is in **Tensorflow** for the *training* and *eval* modality, only in *eval* for the **Tensorflow Lite** implementation. Both implementations can work on CPU and GPUs enabled hardware.

5.2 Time-series RC-ESN

The Time-series RC-ESN operates similarly w.r.t. the Time-series RNN LM but is implemented differently. In particular it uses Echo State Networks (a particular family of Reservoir computing techniques) that are generally more efficient and indicate than RNN for constrained edge devices.

5.2.1 Execution modes

As for the Time-series RNN LM , this LM has two execution modes:

- *Training*: modality to train the underlying prediction model.
- *Eval*: modality to use the prediction model in inference only.

5.2.2 Hyperparameters

At setup (set):

- *num_recurrent_units_per_layer: int; constraints: x > 0; default: 100*

[10] https://numpy.org

- *num_layers: int; constraints: x > 0; default: 1*
- *input_scaling: float; constraints: x > 0; default: 1.0*
- *recurrent_scaling: float; constraints: x > 0; default: 1.0*
- *bias_scaling: float; constraints: x > 0; default: 1.0*
- *leaking_rate: float; constraints: 0 < x <= 1; default: 1.0*

At runtime (set/get)

5.2.3 Input and output

The LM expects the data to be formatted as Numpy tensor as follows:

- **Input**: tensor of shape (*batch_size, sequence_length, input_features_size*)
- **Target**: tensor of shape (*batch_size, sequence_length, target_size*). If *sequence_length* is None, it is assumed that the target refers to the whole sequence (typical case for sequence classification). Otherwise, *sequence_length* must be equal to the corresponding dimension in the input: in this case, to each time step corresponds a target.
- **Target_mask**: optional mask of shape (sequence_length) that can be used to exclude some time steps from the target tensor. This is required for time-series for which only partial ground-truth data is available.

The LM outputs the training loss in the training mode (after each call), while it returns also the predicted tensors (*batch_size, target_size*) for the eval mode.

5.2.4 List of API calls

Here we list the main API of the LM:

- ***init****(network_hyperparams, optimizer_hyperparams, return_sequences=False, return_state=False, masking_value=None, optimizer?)*
  - *Network_hyperparams: num layers, architecture (e.g., gated), readout type, …*
  - *Masking_value: if not None, it is the value used for masking the target tensors*

- ***train****(input_tensor, target_tensor, initial_state?, stateful)*
  - *stateful (bool): whether the network should maintain an internal state across calls (True), or instead the state is reset at each call (False).*
  - *optimizer: Optimizer object, or string "classification"/"regression"*

- ***eval****(input_tensor, initial_state?, stateful) -> predicted tensor*

5.2.5 Implementations of the LM

The implementation of this module is in **Tensorflow** for the *training* and *eval* modality.
If the readout layer is linear and the optimizer is a closed-form optimizer, training is supported also in **Tensorflow Lite**. Both implementations can work on CPU and GPUs enabled hardware.

5.3 Federated Learning

This learning module provides the core functionalities for distributed and federated learning with both edge and server nodes functionalities. More about the usage of this LM can be found in the Federated Learning use-case mockup of Section 6.3.4. If instantiated as an edge node this LM expects to receive an initialization model, updates it locally and re-sends it on the shared MQTT data broker. Otherwise, if implemented as a server node, waits for locally trained models, merge them and re-sends them onto the network.

5.3.1 Execution modes

This LM has two execution modes:

- *node*: main modality for the local updating of the shared model on the edge.
- *Server:* if instantiated in server mode, the federated learning LM acts as centralized server in charge of fusing and distributing trained models.

5.3.2 Input and output

The LM expects the data to be formatted as Numpy tensor data of the following format (*batch_size*, *input_features_size*, *target_size*). If *target_size* equals None, the problem will be assumed as a **sequence learning** problem, a **sequence classification** problem otherwise.

5.3.3 List of API calls

Here we list the main API of the LM:

- ***init****(hyper_params) -> State*
- ***train****(input_tensor) -> State*
- ***eval****(input_tensor) -> predicted tensor*
- ***add_model****(trained_model) -> State*
- ***merge_models****() -> State*
- ***get_merged_model****() -> model*

5.3.4 Implementations of the LM

The implementation of this module is in **Tensorflow** for the *training* and *eval* modality, only in *eval* for the **Tensorflow Lite** implementation. Both implementations can work on CPU and GPUs enabled hardware.

5.4 Continual Learning

This support learning module provides basic utilities for continual learning that can be used by the other learning modules. This module is focused on offering basic *replay mechanisms* that can be used agnostically by almost any other learning algorithm.

5.4.1 Execution modes

This LM has one execution modes:

- *External Memory Management*: modality to handle an external memory buffer to be used for replay.

5.4.2 Input and output

The LM expects the data to be formatted as Numpy tensor data of the following format (*batch_size*, *input_features_size*, *target_size*). If *target_size* equals None, the problem will be assumed as a **sequence learning** problem, a **sequence classification** problem otherwise.

5.4.3 List of API calls

Here we list the main API of the LM:

- ***init****(ext_mem_size) -> State*
- ***update_memory****(input_tensor) -> State*
- ***get_memory_buffer****() -> tensor_buffer*

5.4.4 Implementations of the LM

The implementation of this module is in **Tensorflow** for the *training* and *eval* modality, only in *eval* for the **Tensorflow Lite** implementation. Both implementations can work on CPU and GPUs enabled hardware.

5.5 Privacy-preserving

Training routines for *Privacy-aware Neural Networks*. Privacy-aware training algorithms keep track of the privacy budget during training and modify the model's updates to guarantee the privacy.

5.5.1 Execution modes

This LM has two execution modes:

- *Training*: private training of a specified architecture given a dataset.
- *Eval*: may not be needed if you use other modules at evaluation time.

5.5.2 Input and output

The LM expects the data in the same format as Section 5.1.

The LM expects the data to be formatted as Numpy tensor data of the following format (*batch_size*, *input_features_size*, *target_size*). If *target_size* equals None, the problem will be assumed as a **sequence learning** problem, a **sequence classification** problem otherwise.

The LM outputs the success or error states in the training mode (after each call), while it returns also the predicted tensors (*batch_size*, *target_size*) for the eval mode.

5.5.3 List of API calls

Here we list the main API of the LM:

- ***init****(n_layers, n_neurons, optimizer_hyperparams) -> State*
  same arguments as 4.2, plus privacy-specific hyperaparameters: noise_multiplier, l2_norm_clip, microbatches

- ***train****(input_tensor, privacy_budget) -> loss, privacy_budget*
  privacy_budget is a tuple <epsilon, delta> corresponding to the model's DP coefficients

- ***eval****(input_tensor) -> predicted tensor*

5.5.4 Implementations of the LM

The implementation of this module is in **Tensorflow** for the *training* modality. *Privacy-aware training is not supported by Tensorflow lite* (possible future work). Eval may be supported also for Tensorflow-Lite (and any other architectures supported by the AI Framework). Both implementations can work on CPU and GPUs enabled hardware.

5.6 Dependable AI – Adversarial Robustness

Dependability routine that evaluates the adversarial robustness of a neural network.

5.6.1 Execution modes

This LM has one execution modes:

- *Eval*: given a model a list of samples, find the minimum amount of noise sufficient to perturb the network's output outside the desired bounds.

5.6.2 Input and output

The LM expects the data to be formatted as Numpy tensor data of the following format (*batch_size*, *input_features_size*, *target_size*). If *target_size* equals None, the problem will be assumed as a **sequence learning** problem, a **sequence classification** problem otherwise.
The LM outputs the minimum perturbation necessary to make predictions outside the given bounds.

5.6.3 List of API calls

Here we list the main API of the LM:

- ***init****(d1, d2, d3) -> distances of the three boundaries.*
- ***eval****(model, samples) -> minimum perturbation.*

5.6.4 Implementations of the LM

The implementation of this module is in **Tensorflow** and it supports CPU and GPUs enabled hardware.

5.7 Hyper-parameters Selection

The "*Hyper-parameters Selection"* learning module provides all the utilities to simplify the *hyper-parameters selection* process. In order to automatize "*learning*" in every TEACHING application and provide more transparent API to the end users (i.e. even inexperienced coders), being able to automatically determine the more effective set of hyper-parameters becomes essential. This module offers a simple way to return the best set of parameters given a specific search policy and a set of possible hyperparameters.

Further improvements of this LM may expand current features by providing an even higher-level API that automatically determines those parameters and the search policy based on the specific learning algorithm used and the application context.

5.7.1 Execution modes

This module can be used only for training and as a support LM, i.e. it cannot be directly instantiated in a *routing graph* of an application but should be used directly by another learning module that exploits its features.

5.7.2 Input and output

The inputs to this LM are essentially a tensor "*grid"* containing the set of the possible hyperparameters and a string "*policy*" with the specific search policy to be used.

5.7.3 List of API calls

The list of methods available for this LM are the following:

- ***init****() -> instance of the LM.*
- ***get_parameters****(grid, policy) -> set of best-performing hyper-parameters.*

5.7.4 Implementations of the LM

The implementation will be in pure Python, eventually relying on external Python libraries and the training utilities provided by the other learning modules.

5.8 Anomaly Detection

This module provides basic utilities for training *a Long Short-Term Memory Autoencoder (LSTM-AE)* for anomaly detection tasks. The model learns from normal data instances. Evaluation or inference is applied on mixed data containing both normal and anomalous instances. The module offers access to the underlined trained model functionalities.

5.8.1 Execution modes

This LM has two execution modes:

- *Train*: modality to train the underlying prediction model.
- *Eval*: modality to use the prediction model in inference only.

5.8.2 Input and output

The LM expects the data to be formatted as Pandas dataframe:

- **Input**: a dataframe with normal data of shape (*sequence_length, input_features_size, target_size*) for training, a dataframe with data containing anomalies of shape (*sequence_length, input_features_size*) for evaluation or inference. A *config.py* file for training with a dictionary that contains the following arguments:
    - data.path_normal**:** the path to a dataframe that only contains normal instances.
    - data.path_anomaly: the path to a dataframe that contains both normal and anomalous instances, used in evaluation.
    - data.time_steps: number of timesteps to split the data into subsequences.
    - data.ground_truth_cols: *None* if target columns are not included in the dataframe, otherwise a list with the target's feature names.
    - train.batch_size: batch size for training.
    - train.epochs: training epochs.
    - train.val_subsplits: percentage of data to use for validation.
    - model.storage: path to store the serialized model.

- **Output**: 1) Training mode: A serialized model, and a minmax scaler object. 2) Inference mode: the initial dataframe with an additional 'pred' column with 0 for predicted normal and 1 for predicted anomaly.

5.8.3 List of API calls

Here we list the main API of the LM:

- ***init****(CFG) -> State (CFG is a configuration dictionary that contains the architecture and training parameters)*
- ***load_data****() -> State*
- ***train****() -> State*
- ***eval****() -> predicted tensor*

5.8.4 Implementations of the LM

The implementation of this module is in **Tensorflow** for the *training* and *eval* modality. Currently, both implementations work on CPU enabled hardware.

5.9 Reinforcement Learning

In order to train a Reinforcement Learning model, it is first necessary to decide on the algorithm to use and then define the input, output, states and rewards for the agent. We decided to use the Advantage Actor Critic algorithm (Figure 13), aiming to train an agent that would learn to choose the appropriate driving profile, per driver, based on various input collected through the Sensor API. The actor critic algorithm consists of two networks (the actor and the critic) working together to solve a particular problem. The actor network chooses an action at each time step and the critic network evaluates the quality or the Q-value of a given input state. As the critic network learns which states are better or worse, the actor uses this information to teach the agent to seek out good states and avoid bad states.

The input provided to the RL model at each step is related to the stress level and excitement level of the driver, the state of the road (e.g., taken from a camera sensor) and the state of the car as given by the accelerometers and other sensors. For the RL model it is also necessary to define the set of possible actions and quantify the reward resulting from the action. For this purpose, we define three driving profiles (conservative, normal and aggressive) which may correspond to different top velocity, acceleration/deceleration and steering limits. We consequently assume that the RL chooses the best profile at each moment based on the current state of the vehicle and the driver (input). Depending on the status of the driver at the next step (i.e., stress and excitement) we compute the reward for the chosen action and proceed to the next training iteration.

Our network has two output layers. The first output layer that corresponds to the Actor neural network provides three output values (Action) and uses Softmax as its activation function. The three values are the probabilities for each possible Action (driving mode) that the model can perform. The second output layer that corresponds to the Critic neural network provides an output, a Value that is the sum of all expected future rewards. During model training we employ the output of the actor model and the action with the highest probability is chosen each time (i.e., becomes the suggested Action). The action selected by the actor model was translated by the system to a choice of driving profile that is forwarded to the DMU (at inference time).

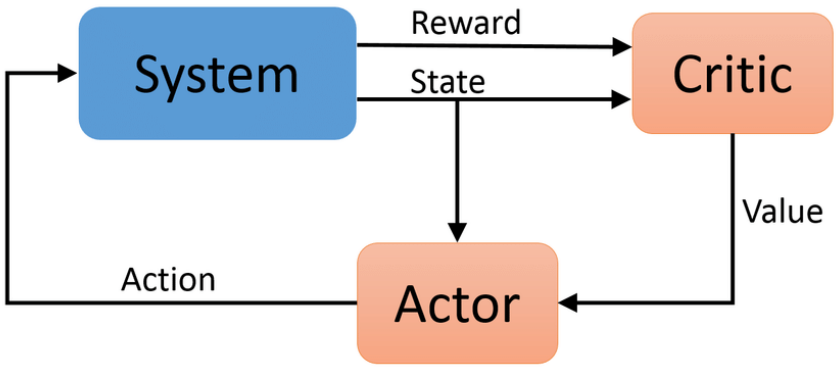


Figure 13 A high-level demonstration of the flow of state observations and reward signals between the algorithm and the environment in the Actor Critic RL architecture.

At each step, a reward was generated for the action taken (the chosen profile), and this reward is computed using a reward function that jointly examines the stress and excitement values of

the driver. The aim is to keep driver's stress at low levels and driver's excitement at high levels. After applying the driving profile, a new package of observations was retrieved from the environment, in order to feed the model again, and the same process is repeated continuously. In parallel, after each execution step, the observations, actions and rewards are kept in a buffer until the requested batch size (500 samples) is covered. The batch is then used for model training. The model was built in TensorFlow v2.0 with Keras v2.0. Two loss functions were also used, one for each model and another for backpropagation. For the actor's neuron the loss function was the product of the logarithm of the probability of the action, which has been selected earlier during the data collection stage, times the difference between the total reward and the reward of the action. Huber loss was used for the critic's neuron. In backpropagation, the sum of the total values of the loss function from each network was used as the loss function. Finally, the Adam optimizer was used with a learning rate of 1e-4. The architecture of the models used is shown in the diagram below (Figure 14).

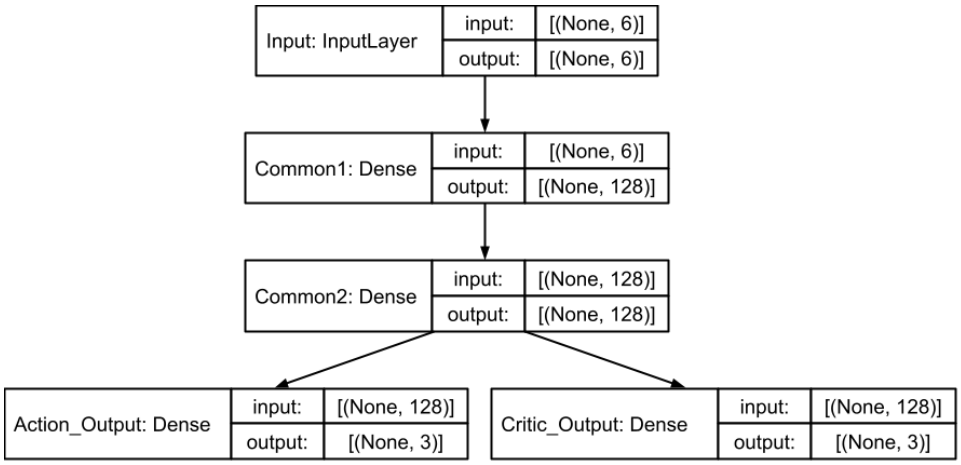


Figure 14 The architecture of the developed RL models.

5.9.1 Execution modes

- *Eval*: given a set of parameters expected as an input by the model, predicted the appropriate driving profile that fits driver's state (stress & boredom) and vehicle status.

5.9.2 Input and output

The LM expects the data to be formatted as follows:

1. **Input**: A set of state measurements that comprise vehicle sensor records, as well as the estimated user stress and boredom levels. These input parameters include:
   - 1.1 y_acceleration: The vehicle acceleration on the y-axis
   - 1.2 gyro_z: The angular velocity on z-axis
   - 1.3 velocity: Vehicle's velocity ox x-axis
   - 1.4 speed_limit: The speed limit as it is defined in the vehicle's environment (i.e. by traffic signs)
   - 1.5 stress: Driver's stress as it comes from the stress recognition module
   - 1.6 boredom: Driver's estimated boredom based on the current driving style

2. **Output**: The predicted driving mode that best matches the driver's style as predicted based on the above factors.

5.9.3 List of API calls

Here we list the main API of the LM:

- ***init****(self, inference_mode , num_inputs , num_actions , steps_per_episode , gamma, num_hidden , learning_rate , max_finish_score)*
- ***load****(pretrained_model_h5_file)*
- ***save****(h5_filename_to_be_saved)*
- ***eval****(model, states) -> preferred driving mode.*

5.9.4 Implementations of the LM

The implementation of this module is in **Tensorflow** for the *eval* modality. The training modality has been implemented and tested in simulation environment only (Carla) since the sensor infrastructure is not yet fully deployed. Both implementations can work on CPU and GPUs enabled hardware.

6 AIaaS Integration

This section is dedicated to illustrating the integration of the AIaaS mockup. In particular, Section 6.1 presents the main software organization, Section 6.2 describes the dependencies and the mockup integration, Section 6.3 describes some use cases, and finally Section 6.4 two demo applications that are fully implemented in Python. We will see how an application developer can declare an application that uses the TEACHING AIaaS framework by providing a few instructions in a Python file in a fast and simple fashion even to solve complex AI tasks.

The reference coding project for the "Artificial Intelligence as a Service (AIaaS) software infrastructure for CPSoS" WP4 of the TEACHING project is the **AI-Toolkit**[11]. The AI-Toolkit is a set of code utilities and services at the core of the AIaaS system. It has been mainly tested, installed, and used on macOS, Linux distributions and on Jetson Nano.

6.1 AI-Toolkit Organization

This subsection is focused on the organization and the structure of the toolkit. The code repository is a private GitLab instance for the TEACHING project. GitLab is a web-based DevOps lifecycle tool that provides a Git repository manager providing wiki, issue-tracking and continuous integration, and deployment pipeline features, using an open-source license, developed by GitLab Inc.

The current project structure visible in the GitLab repository is presented below with a brief description of each main module:

- **teaching**: Main AIaaS package with key learning modules and components. The structure of this module is the following:

```
| teaching
  |- components
      |- ai_framework
      |- application_runtime
      |- application_translator
      |- data_brokering
      |- dmu
      |- ext_common_interface
      |- local_storage
      |- sensor_api
  |- learning_modules
      |- continual_learning
      |- cybersecurity
      |- federeted_learning
      |- hyperparams_selection
      |- privacy_preserving
      |- reinforcement_learning
      |- reservoir_computing_esn
      |- rnn
      |- sum
      lm_base.py
```

[11] https://teaching-gitlab.di.unipi.it/v.lomonaco/ai-toolkit/-/tree/master/#ai-toolkit. The repository is currently private and only accessible by the consortium partners. A zip file containing the latest AI-Toolkit software version will be submitted together with this document for completeness.

- **applications**: Applications Demo implemented on top of the mockup. Beware that some of the logic has been hardwired and implemented independently from the true components and learning modules. The structure of this module is:

```
|- applications
      |- driving-mode-personalization
      |- reinforcement learning
      |- sinwave
      |- stress
```

- **tests**: Tests for the integrated mockup and the individual components / learning modules. Below its structure:

```
|- tests
      |- DepUseCase
      |- FederatedLearning-useCase
      |- SequenceClassificationCL
      |- unit_tests
      README.md
      integration.py
```

- **utils**: Additional materials not strictly necessary for the current AIaaS Toolkit, but that might be useful in the future. Below its structure:

```
|utils
      |- esn_lite
      |- pseudocode_ideas
```

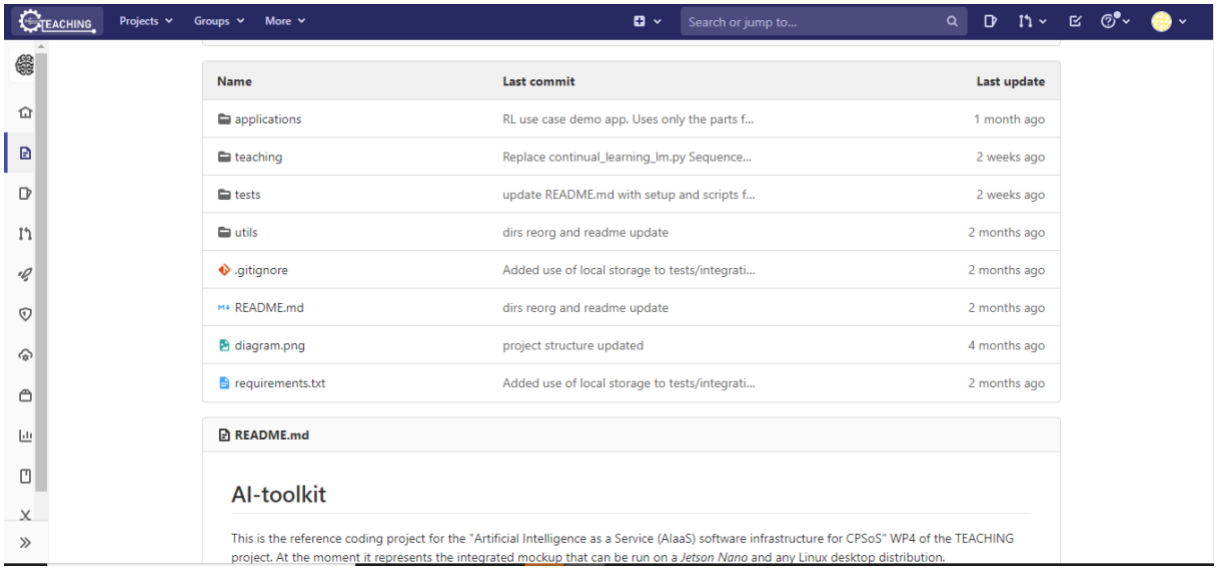


Figure 15 The GitLab AI-Tookit landing page.

6.2 Setup and Mockup Integration Script

This subsection presents everything we need to use the AI-Toolkit and in particular the mock-up integration script. The latter is used to check if all dependencies are installed and in general if the setup was successful. We can divide the setup into four parts called prerequisites,

installation, requirements (or dependencies) and finally the verification via mock-up integration script.

Prerequisites – The main requirements for the execution of the integration script are mainly two:

- Python: It is recommended that you use Python 3.6 or greater, which can be installed either through the Anaconda package manager, Homebrew, or the Python website.

- Package Manager: to install the AI-Toolkit, you will need to use one of two supported package managers: Anaconda or pip. The latter is the recommended package manager as it will provide you in a fast way AI-Toolkit dependencies in one. For Python3 use pip3.

Installation/Clone Repository - Currently, AI-Toolkit source code can be found in GitLab, and it is possible to clone it via SSH[12] or HTTP[13].

Requirements - Here, pip3 is our recommended package manager since it installs all dependencies using the follow command line `pip3 install -r requirements.txt`

Verification - To ensure that the AI-Toolkit works in your system and the dependencies have been installed correctly, it is possible to run the test Python scripts. Such execution represents the integrated mockup that can be run on a *Jetson Nano* and any Linux desktop distribution.

To run the code, launch a test from the project directory in two different terminals that simulate two devises:

```
python3 -m tests.integration
python3 -m teaching.components.sensors_api.virtual_sensors_publisher
```

The Integration script tests the proper working of the interfaces among components and modules. The `virtual_sensors_publisher` is a script implementing 5 virtual sensors, regularly publishing readings to the MQTT broker. Here sensors are eda (sensors/eda). The second script initializes a new application, defines the data sources (sensors/eda), creates local storage, defines the lm and routing like below

```
# Define routing
app.route([
(eda, teaching.output, {}),
(target, lm, {'type': 'label', 'resampling': {'freq': 50, 'buffer_size':
1000}}),  # 'resampling': Hz, 'buffer_size': ms
(eda, lm, {'resampling': {'freq': 50, 'buffer_size': 1}}),
(lm, teaching.output, {})  # Exit point
])
```

The output is simply a continuously updating data of the sensors represented by two plots.

[12] git@teaching-gitlab.di.unipi.it:v.lomonaco/ai-toolkit.git
[13] https://teaching-gitlab.di.unipi.it/v.lomonaco/ai-toolkit.git).

6.3 Mockup Use Cases

In this section, we describe examples of possible use case mockups scenarios that the AIaaS system may enable, focusing (among many others) on the following:

- Sequence Classification with Continual Learning (described in Section 6.3.1)
- Dependability (described in Section 6.3.2)
- Reinforcement Learning (described in Section 6.3.3)
- Federated Learning (described in Section 6.3.4).

The examples have been designed to cover as many AIaaS planned functionalities as possible, as well as covering the main TEACHING use cases. At this point of the project, they are a demonstration of the successful integration of the different AI-Toolkit components, the concrete use of the learning module for practical applications and finally some typical workflows of a generic applications leveraging the proposed platform.

6.3.1 Sequence Classification with Continual Learning

The first use case scenario is sequence classification with or without the *Continual Learning* integration, to show the usability of both learning modules in the AIaaS system.

The goal of this use case is to use the RNN-RC learning module for a stress recognition demo application using the *Continual Learning* support LM for the external memory management, i.e. the basic functionalities needed to handle an external memory buffer to be used for *experience replay*, a basic technique to reduce forgetting in neural networks as discussed in Section 2.3. However, for the sake of this first mockup implementation, a pretrained ESN is used only for inference. Note that the main difference with respect to to the application described in Section 6.4.1 is the use of the support CL learning module to enable ESN model adaptation on non-stationary data streams.

To run the code, launch SequenceClassificationCL from the project directory in two different terminals that simulate two devises.

```
python3 -m tests. SequenceClassificationCL.scCL
python3 -m teaching.components.sensors_api.virtual_sensors_publisher
```

Parts of the code that implements the use case application are shown below. The general schema is to initialize a new application and define the data sources, then define the components, create local storage and define routing graph as follows:

```
# Define routing
app.route([
(eda, teaching.output, {}),
(target, lm, {'type': 'label', 'resampling': {'freq': 50, 'buffer_size':
1000}}),  # 'resampling': Hz, 'buffer_size': ms
(eda, lm, {'resampling': {'freq': 50, 'buffer_size': 1}}),
(lm, teaching.output, {})
# Exit point])
```

Here, we have an additional virtual sensor named "*target*" which emits target labels for the task. By annotating the (target)->(lm) edge with type=label, we inform the framework that the data

flowing through that edge should be used for supervised learning. Within the function `tick()` in Data Bus module, the framework can have access to the labels.

After the runtime of the application is started by calling `asyncio.create_task(app.run())`, the two results can be obtained by calling `app.output()`. As time goes on, the results are accumulated into buffers which are emptied as soon as a new output is requested by calling `app.output()` as presented in Section 6.4.1.

This use case shows the possibility to run the applications of sequence classification even with the presence of the CL module.

6.3.2 Dependability

The second use case choice is for the dependability scenarios. We decided to use the adversarial robustness method and ESN as the neural network model. The purpose of this use-case is to show the possibility of using the Dependability LM and pass not only data from the sensors, but also NN models by passing them through the data brokering module.

To run the code, launch *DepUseCase* from the project directory in two different terminals that simulate two devises:

```
python3 -m tests.DepUseCase.dep_pub  pla
python3 -m tests.DepUseCase.dep_sub
```

Parts of the code that implements the use case application are shown below. The general schema is to initialize a new application, define the component and set some application parameters. For the sake of the mockup we limit ourselves to a simpler version of the scenario, in the case in which only a new model arrives and its robustness is evaluated. The user can simply execute the application using the following code:

```
if __name__ == "__main__":
      loop = asyncio.get_event_loop()
      result = loop.run_until_complete(main())
```

with the *main* code being:

```
async def main():
      loop = asyncio.get_event_loop()
      loop.create_task(app.run())
```

With this use case demo, we illustrate the possibility to use the Dependability LM and evaluate arbitrary NN models such as ESNs.

6.3.3 Reinforcement Learning

In order to showcase the usability of the Reinforcement Learning LM in the context of the AIaaS framework, we implemented a use-case scenario through a demo application that demonstrates its usage. In this use case, the goal is to use a pretrained RL model in the inference stage, feed the model with a set of 6 sensor measurements from an autonomous vehicle and the driver (acceleration on the y axis, gyroscope, velocity, speed limit, excitement level of the

driver, electrodermal activity of the driver) and get the most suitable driving profile of the autonomous vehicle (cautious, normal, sport) as the predicted output of the model.

Parts of the code that implements the use case application are shown below. The following code block depicts the instantiation on the sensor sources and the routing of these data sources to the RL LM.

```
y_acceleration = MqttSensorDevice("sensors/carla/acceleration_y")
app.add_data_source(device=y_acceleration)
#
gyro_z = MqttSensorDevice("sensors/carla/gyro_z")
app.add_data_source(device=gyro_z)
#
velocity = MqttSensorDevice("sensors/carla/velocity")
app.add_data_source(device=velocity)
#
speed_limit = MqttSensorDevice("sensors/carla/speed_limit")
app.add_data_source(device=speed_limit)
#
excitement_sensor = ExcitementSensorDevice()
app.add_data_source(device=excitement_sensor)
#
eda = WesadSensorDevice()
app.add_data_source(device=eda)
```

After creating the RL model instance we route the previous sensor sources into the learning module and the output of the model to the output of our application.

```
rlmodel = RL_Model()
rlmodel.load(path='data/models/RL_model_episode_470.h5')
app.add_module('rl', rlmodel)


app.route([
    (y_acceleration, rlmodel, {}),
    (gyro_z, rlmodel, {}),
    (velocity, rlmodel, {}),
    (speed_limit, rlmodel, {}),
    (excitement_sensor, rlmodel, {}),
    (eda, rlmodel, {}),
    (rlmodel, teaching.output, {})  # Exit point
])
```

And finally, we get the predicted driving mode as an output of the application:

```
rlmodel_output = await app.output()
print('RL LM output: Driving mode {}'.format(rlmodel_output[0]))
```

This output in this use case scenario is not forwarded to any other module or component since this use case only focused on demonstrating the functionality of the RL module but in a more complex application, like the one demonstrated in the section that follows (Section 6.4), this

output could be routed to another LM, a Decision Making Unit that implements any application specific logic etc.

6.3.4 Federated Learning

As mentioned in Section 2.2, Federated Learning enables distributed devices to collaboratively learn a shared prediction model while keeping all the training data on the device, decoupling the ability to do machine learning from the need to store the data in the cloud. TEACHING apps using our AIaaS can leverage the Federated Learning LM to implement it in a federation of hardware agnostic devices.

To show the use of the Federated Learning LM we developed a simple use case with one server and two clients. For simplicity a pre-trained ESN model is used. However, please note that the Federated Learning LM is agnostic to the specific ML model used.

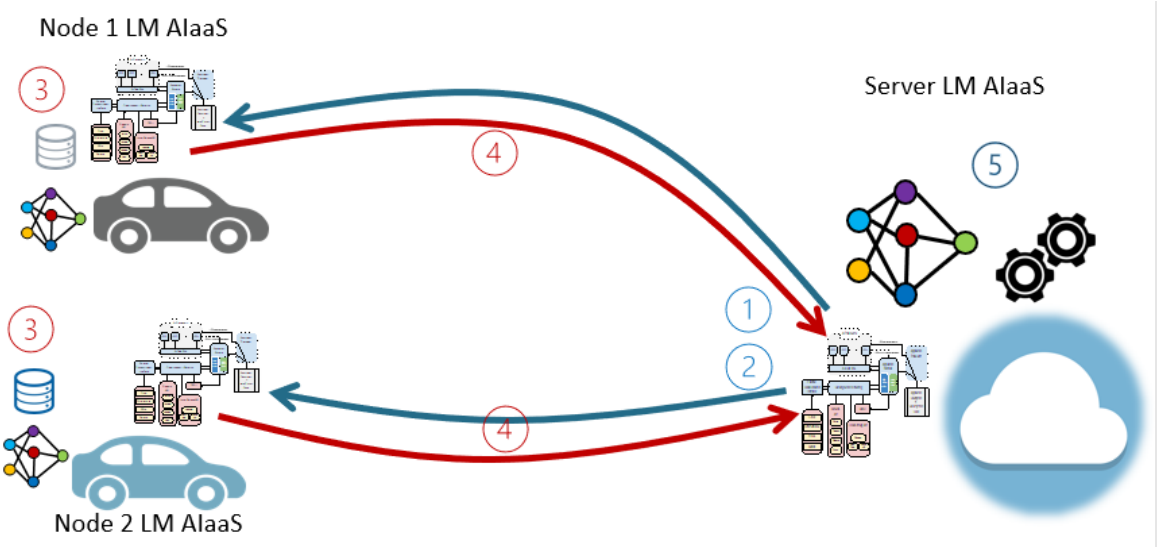


Figure 16 A graphical view of the Federated Learning Use Case

A graphic view is shown in Figure 16. The server initializes the global model (1) and passes it to clients (2). Every client trains the model with local data (3) and then sends the new local model to the server (4). Then the server aggregates the two models (5) and sends the new global model.

All this work is transparent to the user in our framework and needs only to initialize a new application, define the component, set a model, and add LM to the app object. Please note that the user can setup the server node simply as an instance of the AIaaS system, just by instructing correctly the Federated Learning module about it specific execution mode (as a server or as a worker) as shown below:

```
"Setting for user"
app = teaching.init()
# Init a new application
app.metadata = True
# model no data from sensor!
```

```
app.param['use_case'] = 'fl'
app.param['device'] = 'nodes'
# set a NN model and add LM in appesn_
pub = ESN(os.path.realpath(os.path.join(__file__,
'../../../applications/stress/','data/models/esn_stress_recognition.pkl')))
node = NodeFL(esn_pub)
app.add_module(app.moduleName, node)
```

The Application module routes the workflow of the user's application and the Data Brokering module manages the communication between the different devices using MQTT protocol behind. To run the code, launch a *FederatedLearning-useCase* code from the project directory in three different terminals (two as nodes - with one argument as input n1 or n2 - and one as server):

```
python3 -m tests.FederatedLearning-useCase.fl_nodes n1
python3 -m tests.FederatedLearning-useCase.fl_nodes n2
python3 -m tests.FederatedLearning-useCase.fl_server
```

The FL demo use case demonstrates how simple it is to build real-time AI applications by using the AIaaS framework even with complex remote infrastructure and learning approaches.

6.4 Demo Applications

In this section we describe two applications leveraging the AIaaS framework. The integration script is needed to check if the setup was done correctly while the mock-ups highlight the possible use-cases and general workflow of an application. Here the focus is on two specific uses of the AIaaS system for two demo applications very relevant to TEACHING. Section 6.4.1 illustrates the stress recognition demo app, while Section 6.4.2 shows the autonomous driving personalization module.

6.4.1 Stress Monitoring

The stress recognition demo application showcases the ability of the AIaaS framework to easily allow the construction of applications by interactions of basic components.
In the stress recognition demo application, the goal is to recognize the level of stress of a user from physiological sensors. For the purposes of this demo, we limit ourselves to the use of an electrodermal activity (EDA) sensor for input. The output is simply a continuously updating measure of the current level of stress.

The actual data used for the demo does not come from a real EDA sensor. Instead, we have used the data included in a publicly available dataset in the literature (WESAD) in order to simulate a real-time stream. We refer to the producer of this data as a virtual sensor.

The application uses a pretrained Echo State Network for inference, so the main components of the applications are the input sensor (eda) and the Echo State Network (lm). Given these two components, the framework allows the definition of a stress recognition application simply by declaring a routing graph:

```
app.route([
    (eda, teaching.output, {}),
    (eda, lm, {}),
```

```
    (lm, teaching.output, {})
])
```

Each item in the routing list defines an edge in a computational graph. In this case, the data emitted by the sensor is fed to the `lm` (i.e., to the ESN) and to the output of the application (`teaching.output`) for visualization. The output of the ESN is fed directly to the output of the application.

After the runtime of the application is started by calling `asyncio.create_task(app.run())`, the two results can be obtained by calling `app.output()`. As time goes on, the results are accumulated into buffers which are emptied as soon as a new output is requested by calling `app.output()`.

The stress recognition demo application demonstrates how simple it is to build real-time AI applications by using the AIaaS framework.

6.4.2 Autonomous Driving Personalization

We decided to use the Reinforcement Learning method with the Advantage Actor Critic algorithm, aiming to train an agent that would learn to choose the appropriate driving profile, per driver, based on his stress and excitement values. To do this, we used an ANN model to simulate the stress and excitement values based on the vehicle condition, and we modified CARLA's Behavior Agent. The Behavior Agent has been modified in such a way that dynamically during the execution of a route one can change the driving profile between three options: conservative, normal and aggressive.

To apply the Reinforcement Learning method to our experiments, we made the necessary modifications to CARLA 's execution script. A new script was created, which placed the vehicle randomly in CARLA's map and at the same time activated the autopilot giving it a random route. The same script repeated the same process each time the experiment was reset. At the same time, in order to simulate the different drivers, in each reset of the experiment, the initial profile of the Behavior Agent was randomly selected, but at the same time the driver profile was related to how anxious and excited he was depending on the condition of the vehicle. This way, we were able to give the agent as many new scenarios as possible with different drivers and different routes so that he could gain a good experience of the world.

In order to train the RL model for the driving personalisation task, we employed the CARLA driving simulator environment (https://carla.org/), which allowed us to instantiate an autonomous vehicle that is operated by the CARLA parametric auto-pilot (Figure 17). The parametric auto-pilot allows to set the various driving parameters (e.g., maximum speed, maximum steering angle, acceleration or deceleration etc) and achieve different driving profiles that correspond to different autonomous vehicle driving modes. In addition, we are able to simulate several vehicle sensors that measure the speed, acceleration and rotation of the vehicle on all axes (which implicitly affects the passengers' stress), and use them as input to the RL.

Figure 17 The autonomous vehicle used in the Carla simulator, operated by the parametric auto-pilot that was deployed for testing the autonomous driving personalization algorithms and developing the relative demo application to showcase the results.

This way we were able to give the agent as many new scenarios as possible with different guides and different routes so that he could gain a good experience of the world. The condition for resetting the experiment was to complete a path, fail to complete it, or collect enough data for the batch. The sensors selected to be given to the model were the horizontal acceleration y, the vertical acceleration x, the angular velocity on the z-axis, the vehicle speed, the speed limit of the road moving the vehicle as well as the ***stress*** and ***excitement*** values. The choice of these sensors arose from the experiments of previous chapters and it was decided that they give a very good picture of the current state of the world and the respective driver.

During the training phase, 500 observation batches were created. The observations are used to calculate loss functions and update weights. Finding the right reward function was a challenge since a wrong choice may result in strange model behaviours (e.g., always selecting a profile that ***minimizes stress*** independently of the ***excitement*** or the inverse). Finding the right learning rate was also difficult. With each unsuccessful fine-tuning experiment taking 4 to 5 hours and the final successful train of the model lasting 24 hours, we finally got a model that switches driving profiles during driving taking into account the stress and excitement in tandem. The model has been trained on an AMD Ryzen 9 5900X CPU (12 cores) with 32 GB RAM and an RTX 3070 OC 8GB graphics card. The most significant delay in the experiments was due to the need to make all training epochs in real time with Carla's virtual world, so the training hours correspond to real driving hours. In total the model was trained for about 900 episodes and the training process was interrupted when it was considered that the model did not improve further. The final model chosen is that of 470 episodes.

In order to evaluate our model, we employed the CARLA autopilot as a baseline and the stable choice of a driving profile during the whole route. We also compared against a method that randomly changes driving profiles during the route. We repeated the experiment for 5 random routes and the results show that with the proposed method we had a decrease in stress between 4 and 15% and an increase between 1 and 15% in excitement.

With respect to the above experiments and the framework units as described in Section 3 and 5, we also developed a TEACHING demo application that demonstrates the integration of various parts of the AIaaS platform. For this purpose, the CARLA driving simulation environment has been installed and configured in order to allow integration with the TEACHING framework. In essence, this application uses the communication among the different components to flow the extracted user stress levels coming from the stress recognition module to feed the RL module along with various vehicle sensor data to predict the appropriate autonomous vehicle driving profile that should be used by the vehicle's agent for personalizing its behaviour based on the user's predicted profile. In addition to the above, and in order to facilitate the demo app implementation and test the whole TEACHING pipeline, the vehicle sensors from CARLA have been wrapped up and delivered through a message brokering service, a module that simulates user stress and user excitement based on the vehicle behavior (speed, acceleration, etc.) has been developed in order to provide input for the RL module and allow to train the RL model.

An example of the routing used in this demo application to flow the information among the various components and learning modules is given below.

```
app.route([
    (eda, stresslm, {}),
    (eda, teaching.output, {}),
    (stresslm, teaching.output, {}),
    (stresslm, rlmodel, {}),
    (excitement_sensor, rlmodel, {}),
    (y_acceleration, teaching.output, {}),
    (y_acceleration, rlmodel, {}),
    (gyro_z, rlmodel, {}),
    (velocity, rlmodel, {}),
    (speed_limit, rlmodel, {}),
    (rlmodel, teaching.output, {})
])
```

Based on the output of the RL model that comes from the Teaching output node, we use the DMU component to pass the appropriate driving profile change as a command to the autonomous vehicle, using the publish_to_topic function of the DMU that publishes the command to the same broker topic that the vehicle controller is listening for commands.

```
def publish_to_topic(self, topic, value):
        try:
            ret1 = self.client.publish(topic, value)
            print(ret1)
        except Exception as e:
            raise Exception("Queue {topic} doesn't exist. Get
getActiveProviderList for a complete list")
```

A demo video of the application developed that utilizes the Teaching AIaaS Toolkit is available online.[14]

[14] https://www.youtube.com/watch?v=xcK9E6d7CUM

7 Further Preliminary Results

As a part of the activities of WP4, several lines of scientific investigation focused on relevant ML methodologies are currently active, in an exploratory or preliminary phase. The major outcomes of this ongoing and preliminary work (spanning all active tasks T4.1, T4.2, T4.3 and T4.4) are reported in this section.

More specifically, we show in Section 7.1 how to train *ESNs with Tensorflow Lite*. In Section 7.2 we show our preliminary results on *Adversarial Robustness of Recurrent Models*, a fundamental component of our proposal to ensure safety and dependability of recurrent networks. In Section 7.3, we show our preliminary results on *Anomaly Detection with ESNs*, which will be fundamental for the anomaly detection in the avionics use case. Finally, we have started to experiment with *Continual Learning with ESNs*. We show our experimental results in Section 7.4, specifically for *Human State Monitoring* tasks.

7.1 Training ESNs with Tensorflow Lite

Training RNNs "on-the-edge" requires efficient algorithms and lightweight libraries able to run on low-powered devices. For this purpose, we explored the use of TensorFlow Lite (TF-LITE), a deep learning library designed with efficiency in mind. Unfortunately, TF-LITE does not offer support for training natively, and therefore we had to implement the algorithms by ourselves. The work done here may be useful in the future to efficiently deploy recurrent models that are also trained on the edge. Furthermore, training on-the-edge allows to guarantee the user's privacy since the data never leaves the vehicle.

In these scenarios, we focused on ESNs, due to their efficient training algorithms. The model is a basic Keras model, that will be converted into an appropriate format by TF-LITE. To train the ESN we evaluated both a ridge regression and a direct solution computed using the pseudoinverse. It is important to notice that not all the TensorFlow functions are supported by TF-LITE. The basic TF-LITE runtime environment may limit the choice of possible algorithms or require to reimplement some functionality if not available. For example, TF-LITE does not support sparse linear algebra operations, which would have been useful for our implementation. As an alternative, we used the corresponding operations on dense matrices. Similarly, the pseudoinverse computation is supported by TensorFlow but not TF-LITE.

7.1.1 Implementation

The basic idea behind the implementation is that training an ESN is very similar to transfer learning, where we keep the feature extractor fixed and only update the final classifier.

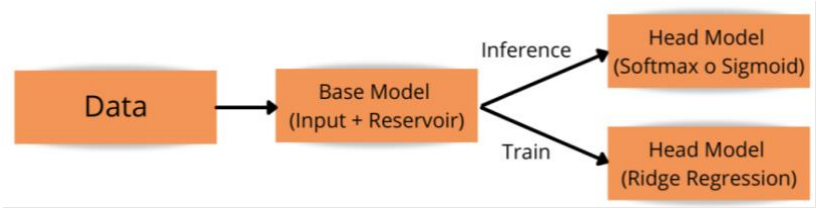


Figure 18 ESN separated into Base Model and Head

As a result, our model is divided into two components: the base model, comprising the input layer and recurrent reservoir, and a final head, which performs the classification using the reservoir's features (Figure 18). Notice from the figure that we have two heads, one for training, and another for inference. This is necessary because these components are implemented via TF-LITE Concrete Functions, and because TF-LITE does not allow to update the parameters.

- **Inference Head** performs the classification by doing a matrix multiplication followed by a softmax activation.
- **Training Head** is the training algorithm computing the updated parameters.

After the training head computes the updated parameters, the weights can be saved in the disk and the concrete function must be initialized again using the updated parameters. Instead, during inference the inference head outputs the class probabilities for the input sequences.

In TF-LITE, concrete functions represent a computational graph, similarly to TensorFlow. The conversion from a TensorFlow computational graph to a TF-LITE concrete function results in a series of optimizations that makes the resulting computation more efficient, for example by applying model quantization or fusing operations together.

Additional flags can be activated during the conversion process to enable operations which are normally unavailable, such as the matrix inverse that we need to implement the ridge regression. After the conversion, the concrete function can be saved and loaded on-device. In our experiments, we focus on Python, but it is important to notice that TF-LITE supports many other languages. We can load a concrete function using a TF-LITE interpreter, and performs its computations by calling the method *invoke*, using the concrete function and input as arguments.

7.1.2 Results

We compared the efficiency of TF-LITE against TensorFlow for the base model (ESN reservoir), training function (training head) and inference (inference head). We compared the CPU time (Table 7), disk space (Table 8), and RAM usage (Table 9).

Table 7 Comparison of CPU time of TensorFlow against TF-LITE

Model	CPU times TFL	Wall Time TFL	Exec Time TFL per loop	CPU times TF	Wall Time TF	Exec. Time TF per loop
ReservoirModel (single-input)	3.35 ms	2.58 ms	2.14 ms ± 99.7 μs	69.1 ms	42.6 ms	43.2 ms ± 4.31 ms
ReservoirModel (multiple-input)	65.8 ms	64.6 ms	45.8 ms ± 1.33 ms	209 ms	84 ms	82.7 ms ± 1.95 ms
InferenceModel	505 μs	485 μs	55.9 μs ± 885 ns	1.02 ms	759 μs	508 μs ± 46.6 μs
TrainModel	2.74 ms	1.51 ms	627 μs ± 25.6 μs	3.13 ms	2.27 ms	1.26 ms ± 30.5 μs

Table 8 Disk space of TensorFlow and TF-LITE

Model	Disk Space TF	Disk Space TFL
ReservoirModel	1,6MB	91,6kB
InferenceModel	19,6kB	1,9kB
TrainModel	24,4kB	4,2kB

Table 9 RAM usage of TensorFlow against TF-LITE

Algorithm	RAM value TF	RAM value TFL
ReservoirAlgorithm (single-input)	366996	347904
ReservoirAlgorithm (multiple-input)	377244	362348
InferenceAlgorithm	367720	350840
TrainAlgorithm	373828	407880

Overall, we can see that TF-LITE can be orders of magnitudes better than TensorFlow in terms of CPU time and disk space used, making it a very promising solution for on-device training. The only disadvantage of TF-LITE is that by default training is not supported. While it is possible to train models with TF-LITE, as we showed in these sections, it is more difficult to implement than using TensorFlow since the programmer is constrained by the limitations imposed by TF-LITE's environment.

7.2 Preliminary Results on Adversarial Robustness of Recurrent Models

As we have seen in Section 2.6, the robustness of RNNs is a fundamental property to determine their safety and proper measures to guarantee dependability.

To give an example of the quantification of robustness, we provide in this section an experimental evaluation of RNNs on WESAD [45], a stress recognition dataset with physiological and motion data from the users. We only provide a very preliminary evaluation. For these reasons, we did not perform a large scale evaluation of recurrent models, and we use hyperparameters which were found optimal from our previous internal experiments. We leave a formal experimental evaluation as future work.

Notice that here we do not consider threshold values d_1, d_2, d_3 defined in Section 2.6.1 and instead we compute the amount of noise necessary to craft an adversarial example. Finally, notice that the robustness value can be hard to interpret by looking at its absolute value. It is better instead to compare different models against each other. For example, in our experiment the average robustness of the models shows that the most robust model, RNN-64, is also the less accurate.

Table 10 shows the results for a vanilla RNN with 64, 128, 256 hidden units, respectively. Each model is trained to optimize the cross-entropy on the training data, using an Adam optimizer for 200 epochs. After each epoch, the current model is evaluated on a validation set, and the best model is restored after the training loop is completed. The results in the table are computed on a separate test set. After training, we compute the prediction's robustness over a set of 50 samples, by finding the amount of noise needed to craft an adversarial example. We use POPQORN [37] to compute such values.

Table 10 Accuracy and adversarial robustness of RNNs trained on WESAD.

	Accuracy	Robustness			
		min	mean	max	std
RNN-64	0.83	0.0014	0.0890	0.2717	0.0717
RNN-128	0.86	0.0008	0.0022	0.0079	0.0014
RNN-256	0.89	0.0031	0.0353	0.1361	0.0334

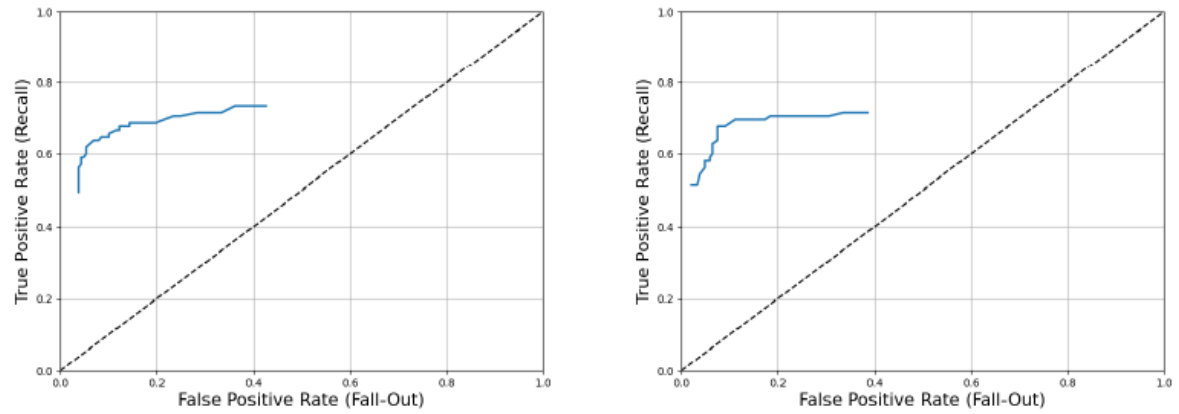
It is worth noticing that we do not consider threshold values d_1, d_2, d_3 defined in Section 2.6. Instead, we compute the amount of noise necessary to craft an adversarial example. Finally, notice that the robustness value can be hard to interpret by looking at its absolute value. It is better instead to compare different models against each other. For example, in our experiment the average robustness of the models shows that the most robust model, RNN-64, is also the less accurate.

7.3 Preliminary Results on Anomaly Detection with Echo State Networks

Due to the complexity of avionics systems, and the complexity of their protocols, modern monitoring systems needs to be constantly aware of the current state of the system, and actively catch any eventual anomaly. The available sensors can be used to collect useful data, both for normal and anomalous situations. These data can be studied to classify anomalous behaviour in the future as soon as it occurs, to avoid disasters. As a preliminary study, we decided to evaluation RNNs and ESNs on anomaly detection on time series datasets.

We used telemetry data from the Soil Moisture Active Passive (SMAP) satellite, and from the Curiosity rover on Mars (MSL). Data have been anonymized, separated into train and test set, and normalized between -1 and +1. The data are separated into multiple channels. Notice that the data has been collected during real accidents, and the label have been created by domain experts.

Currently, we are still working on the experimental phase. Preliminary results are promising and show that both LSTMs and ESNs are able to detect anomalous situations. We show the ROC curves for LSTMs (Figure 19) and ESNs (Figure 20). Table 11 summarizes the results. Unfortunately, we still do not have access to anomalies from the avionics use case, therefore we had to resort to datasets from the literature. We will perform an experimental evaluation on avionics data whenever such data becomes available.



(a) ROC Curve LSTM 1 layer (b) ROC Curve LSTM 2 layers

Figure 19 ROC curves LSTMs.

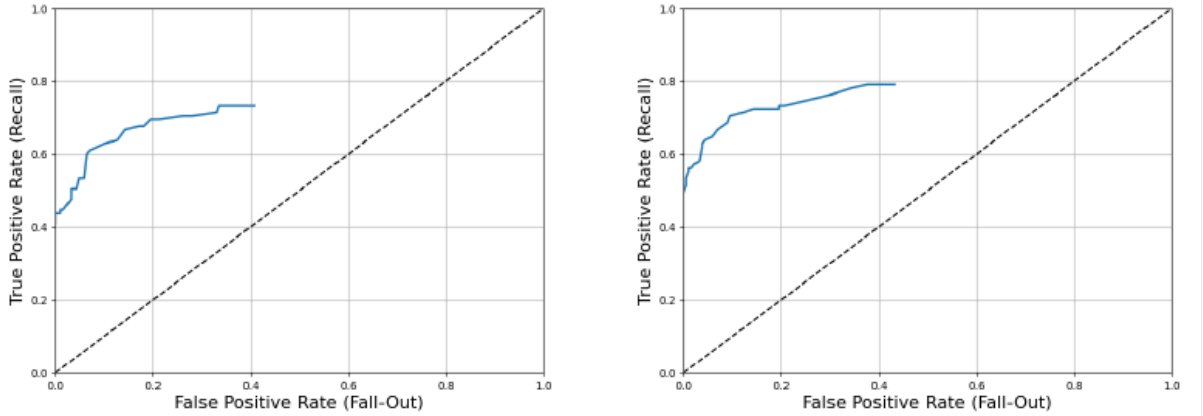


Figure 20 ROC curves ESNs.

Table 11 Results on anomaly detection datasets.

	Final results				
Model	Precision	Recall	True Positives	False Positives	False Negatives
LSTM 1l	0.77	0.64	67	20	38
LSTM 2l	0.68	0.75	79	37	26
ESN 1l	0.76	0.65	68	22	37
ESN 2l	0.82	0.72	76	17	29

7.4 Continual Learning with Echo State Networks

As we have discussed in Section 2.3, learning from a stream of data without forgetting previous knowledge is one of the key challenges in continual learning. However, the use of RNNs introduces additional issues with respect to common feedforward models, since large input sequence lengths reduce the ability of existing continual learning strategies to mitigate forgetting.

We believe that recurrent models like ESNs can be effectively leveraged to mitigate catastrophic forgetting. The fact that ESNs have an untrained recurrent component can benefit the continual learning performance in several ways: first, forgetting cannot originate from parameters that do not change (they are completely stable). Therefore, continual learning strategies do not need to consider them. Second, since training by backpropagation does not require to compute the gradients of recurrent parameters across all input time-steps, ESNs should be able to mitigate the negative effect of large input sequence lengths described above. Third, the fixed ESN component (called reservoir) can be leveraged as a pretrained network. This is a widely used solution in continual learning for computer vision applications, where there is a large availability of pre-trained models. Therefore, ESNs allow to effectively exploit such strategies.

7.4.1 Results

In [46] we studied the aforementioned aspects with experiments on two continual learning benchmarks. In particular, we evaluated the performance of ESNs in terms of catastrophic forgetting on two class-incremental continual learning benchmarks: Split MNIST and Synthetic Speech Commands. The former is a benchmark composed of 5 tasks, each of which takes images from 2 classes of MNIST. Images are taken one row at a time, producing sequences of 28 time-steps. The latter is a benchmark with 10 tasks, each of which presents audio samples representing 2 different spoken words. Each sequence has 101 time-steps. At the end of training on all tasks, we evaluated the model average accuracy across a separate test set of all tasks.
We employed four popular continual learning strategies not specifically designed for recurrent models: EWC, LWF, Replay and SLDA. We also reported the performance for Naive finetuning (training without any continual learning technique) and Joint Training (training on all data at once). They can be considered respectively as a lower and upper bound on the continual learning performance.

Table 12 Average Accuracy across all tasks for ESN and LSTM with popular continual learning strategies. Taken from [46].

SMNIST	LSTM†	ESN
EWC	$0.21_{\pm 0.02}$	$0.20_{\pm 0.00}$
LWF	$0.31_{\pm 0.07}$	$0.47_{\pm 0.07}$
REPLAY	$\mathbf{0.85_{\pm 0.03}}$	$0.74_{\pm 0.03}$
SLDA	—	$\mathbf{0.88_{\pm 0.01}}$
NAIVE	$0.20_{\pm 0.00}$	$0.20_{\pm 0.00}$
JOINT	$0.97_{\pm 0.00}$	$0.97_{\pm 0.01}$

SSC	LSTM†	ESN
EWC	$0.10_{\pm 0.00}$	$0.09_{\pm 0.02}$
LWF	$0.12_{\pm 0.01}$	$0.12_{\pm 0.02}$
REPLAY	$\mathbf{0.74_{\pm 0.07}}$	$0.36_{\pm 0.07}$
SLDA	—	$\mathbf{0.57_{\pm 0.03}}$
NAIVE	$0.10_{\pm 0.00}$	$0.10_{\pm 0.00}$
JOINT	$0.89_{\pm 0.02}$	$0.91_{\pm 0.02}$

The results in Table 12 showed that ESNs behave similar to LSTM networks when trained together with popular continual learning strategies. Replay is the only strategy for which LSTM outperforms ESN. However, additional studies (not yet published) confirmed that ESNs with output feedback connections are able to close the performance gap for replay strategies.
The performance of Deep SLDA is promising on both benchmarks: this is a crucial aspect of the study since SLDA can only be applied in the presence of a fixed feature extractor. Since there is no availability of pretrained LSTM models, ESNs is the only choice to leverage SLDA on these benchmarks.

7.4.2 Discussion

By studying and developing recurrent models able to learn continuously, it will be possible to tackle many continual learning applications which currently lack a working solution: from stock prediction to human activity recognition, the need of recurrent models in dynamic environments is widespread. Additionally, training on the edge will be a key enabler for continual learning, since sending data to a centralized data center requires time and is not always possible (data privacy issues). ESNs promise to combine these two aspects: an efficient, on-device implementation (e.g. through neuromorphic hardware) able to be trained on the edge with fast and efficient continual learning strategies that only have to deal with a linear classifier, instead of a complex recurrent architecture.

7.5 Continual Learning for Human State Monitoring

Having robust Continual Learning algorithms for Human State Monitoring data is critical for real applications. By comparing neural networks trained with continual learning methods to a neural network trained via a classic offline training we can figure out how different the final inference results will be using the continual learning methods. We have selected 6 simple continual learning methods and compared the results to the offline training. The selected methods are the following:

1. *Naive* continual learning, where the data is passed to the training of the neural network without further actions
2. *Replay*, where we keep a percentage of the training data to use as a replay data in the next training session
3. *Cumulative*, a replay method with a percentage of 100%: we bring all the training data to the next training session
4. *Episodic*, in which we keep a fixed number of examples per class and bring them to the next training session
5. *Learning Without Forgetting*, per the homonym paper [47].
6. *Elastic Weight Consolidation*, following the introductory paper [48].

The comparisons were done with data from two popular Human State Monitoring datasets: WESAD and ASCERTAIN. The first one, WESAD, contains data from 15 subjects gathered in a laboratory experiment regarding stress levels: the data were recorded by two devices, one on the wrist and one on the chest, that recorded levels of respiration, body temperature, ECG and more. The second dataset, ASCERTAIN, is very similar, with data from 58 subjects gathered with commercial devices that recorded respiration, skin galvanic responses, ECG and more.

We compared the various methods by measuring the number of epochs needed for training, the time it required, the average accuracy over the training sessions, the final accuracy, forward and backward knowledge transfer and the memory used during the process.

The results (Table 13 and Table 14) show how the data really affects the results of the model. Over WESAD, with a neural network of two GRU layers of 18 units, we got a 99% accuracy with the offline training, and an accuracy above 70% with each of the continual methods with the best performance obtained by the cumulative method which got a final accuracy of 96%. ASCERTAIN showed up to be more of a challenge, with the offline training on a neural network of two GRU layers of 24 units obtaining an accuracy of 42.78% and each continual

learning method staying between 35% and 40%. This proves once again how the continual learning methods are comparable to the offline learning.

Table 13 Results over WESAD

Scenario	Epochs	Time	Accuracy	ACC	BWT	FWT	Memory
Offline	28	94,69s	99,07	-	-	-	2061,40 Mb
Continual	49,14±33,67	947,24s	73,13±4,02	0,7721	0,0343	0,5397	2173 Mb
Cumulative	39,71±19,91	2786s	81,97±8,67	0,961	0,1383	0,4674	2291,45 Mb
Replay	41,14±21,19	1063,51s	78,29±3,32	0,7849	-0,002	0,4582	2184,77 Mb
Episodic	35±26,26	1088,29s	82,13±6,60	0,9095	0,0841	0,4226	2097,47 Mb
EWC	29,71±17,38	1342,81s	70,74±4,74	0,7251	0,0113	0,4698	2187,40 Mb
LWF	44,29±18,30	3282,51s	69,09±5,82	0,7419	0,0451	0,3248	2121,31 Mb

Table 14 Results over ASCERTAIN

Scenario	Epochs	Time	Accuracy	ACC	BWT	FWT	Memory
Offline	3	29,14s	42,78	-	-	-	1817,28 Mb
Continual	10,88±5,01	249,28s	37,45±5,01	0,25	-0,0168	0,0213	2154,36Mb
Cumulative	9,25±6,81	932,56s	39,06±4,26	0,2697	0,0064	0,0278	2297,17 Mb
Replay	13,25±10,03	402,96s	39,48±4,37	0,2603	-0,014	0,0485	2173,95 Mb
Episodic	12,62±7,94	498,24s	38,80±4,04	0,2742	0,0048	0,0156	2231,42 Mb
EWC	24,14±16,94	810,96s	36,08±5,66	0,2497	-0,0183	-0,0003	2171,62 Mb
LWF	21,62±10,20	879,68s	35,69±5,43	0,2448	0,0327	0,0156	2103 Mb

8 Conclusion

WP4 has the role of designing the necessary methodologies and software for creating the TEACHING AI as a Service (AIaaS) system. Currently, all the four tasks of the WP are active.

In this document we have provided a report on the integrated mockup of the AIaaS, providing an overview of the refined architectural design, as well as a detailed description of the platform components and of the identified learning modules. Moreover, we have illustrated the integration process, giving an in-depth description of the integration scripts and demos. We also updated the state-of-the-art analysis and gave preliminary results on on-going AI-related research work. Together with the other deliverables delivered at M20, this document contributes to the fulfillment of the project's Milestone MS2 (First integrated setup with mock-up of the TEACHING platform).

Following the envisaged project's lifecycle, the outcomes of the work conducted in WP4 and described in this document, along with that of the other technical WPs, will be important to complete the core technological building (Phase 2 of the project), following a continuous process of integration with progressively more advanced functionalities (towards Milestone MS4), and to drive the efforts in the use case integration and validation (Phase 3 of the project, towards Deliverable D4.3 and Milestones MS5-6).

9 Bibliography

[1] J. F. Kolen and S. C. Kremer, A field guide to dynamical recurrent networks, John Wiley & Sons, 2001.

[2] H. Jaeger and H. Haas, "Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication," *Science,* vol. 304, pp. 78-80, 2004.

[3] C. Gallicchio and A. Micheli, "Architectural and markovian factors of echo state networks," *Neural Networks,* vol. 24, no. 5, pp. 440-456, 2011.

[4] M. Lukosevicius and H. Jaeger, "Reservoir computing approaches to recurrent neural network training," *Computer Science Review,* pp. 127-149, 2009.

[5] G. Tanaka, T. Yamane, J. B. Héroux, R. Nakane, N. Kanazawa, S. Takeda, H. Numata, D. Nakano and A. Hirose, "Recent advances in physical reservoir computing: A review," *Neural Networks,* 2019.

[6] D. Verstraeten, B. Schrauwen, M. d'Haene and D. Stroobandt, "An experimental unification of reservoir computing methods," *Neural Networks*, pp. 391-403, 2007.

[7] K. Nakajima and I. Fischer, Reservoir Computing, Springer, 2021.

[8] C. Gallicchio, A. Micheli and L. Pedrelli, "Deep reservoir computing: A critical experimental analysis," *Neurocomputing,* vol. 268, pp. 87-99, 2017.

[9] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation,* vol. 9, no. 8, p. 1735–1780, 1997.

[10] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *EMNLP,* 2014.

[11] N. Bertschinger and T. Natschlager, "Real-time computation at the edge of chaos in recurrent neural networks," *Neural Computation,* vol. 16, no. 7, p. 1413–1436, 2004.

[12] B. Schrauwen, M. Wardermann, D. Verstraeten, J. Steil and D. Stroobandt, "Improving reservoirs using intrinsic plasticity," *Neurocomputing,* vol. 71, pp. 1159-1171, 2008.

[13] C. Gallicchio, A. Micheli and L. Silvestri, "Phase Transition adaptation," in *International Joint Conference on Neural Networks (IJCNN),* 2021.

[14] A. Rodan and P. Tino, "Minimum complexity echo state network," *IEEE Transactions on Neural Networks,* vol. 22, no. 1, pp. 131-144, 2010.

[15] T. Li, A. K. Sahu, A. Talwalkar and V. Smith, "Federated Learning: Challenges, Methods, and Future Directions," *IEEE Signal Processing Magazine,* 2020.

[16] B. McMahan, E. Moore, D. Ramage, S. Hampson and B. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*, 2017.

[17] D. Bacciu, D. Di Sarli, P. Faraji, C. Gallicchio and A. Micheli, "Federated Reservoir Computing Neural Networks," *International Joint Conference on Neural Networks*, 2021.

[18] V. Lomonaco, Continual learning with deep architectures, PhD Dissertation, Alma Mater Studiorum - University of Bologna, 2019.

[19] D. Bacciu et al., "TEACHING: Trustworthy Autonomous Cyber-physical Applications through Human-Centred Intelligence," in *IEEE COINS*, 2021.

[20] A. Cossu, A. Carta, V. Lomonaco and D. Bacciu, "Continual Learning for Recurrent Neural Networks: an Empirical Evaluation," 2021.

[21] J. Janai, F. Güney, A. Behl and A. Geiger, "Computer vision for autonomous vehicles: Problems, datasets and state of the art.," *Foundations and Trends® in Computer Graphics and Vision,* pp. 12(1–3), 1-308., 2020.

[22] F. Codevilla, M. Müller, A. López and V. Koltun, "End-to-end driving via conditional imitation learning.," in *In 2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018, May.

[23] M. P. Kebria, A. Khosravi, M. S. Salaken, M. S. Salaken and S. Nahavandi, "Deep imitation learning for autonomous vehicles based on convolutional neural networks," *IEEE/CAA Journal of Automatica Sinica,* pp. 7(1), 82-95., 2019.

[24] E. A. Sallab, M. Abdou, E. Perot and S. YogamanI, "Deep reinforcement learning framework for autonomous driving.," *Electronic Imaging,* pp. 2017(19), 70-76., 2017.

[25] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, ... and K. Zieba, "End to end learning for self-driving cars.," in *arXiv preprint arXiv:1604.07316*, 2016.

[26] S. Elmalaki, R. H. Tsai and M. Srivastava, "Sentio: Driver-in-the-loop forward collision warning using multisample reinforcement learning.," in *In Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, 2018, November.

[27] R. Liu, A. Sarkar, E. Solovey and S. Tschiatschek, "Evaluating Rule-based Programming and ReinforcementLearning for Personalising an Intelligent System.," in *In IUI Workshops*, 2019, January.

[28] Y. A. Gao, W. Barendregt and G. Castellano, "Personalised human-robot co-adaptation in instructional settings using reinforcement learning.," in *In IVA Workshop on Persuasive Embodied Agents for Behavior Change: PEACH 2017*, Sweden, Stockholm, 2017, August 27.

[29] B. Abera, Y. Naudet and H. Panetto, "Towards a Personalisation Framework for Cyber-Physical-Social System (CPSS).," in *In 17th IFAC Symposium on Information Control Problems in Manufacturing.*, 2021, June.

[30] V. Mnih, P. A. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, ... and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning.," in *In International conference on machine learning*, 2016, June.

[31] X. J. Wang, Z. Kurth-Nelson, D. Tirumala, H. Soyer, Z. J. Leibo, R. Munos, ... and M. Botvinick, "Learning to reinforcement learn.," in *arXiv preprint arXiv:1611.05763.*, 2016.

[32] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar and L. Zhang, "Deep Learning with Differential Privacy," in *ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2016.

[33] I. O. f. Standardization, *ISO 26262:2018, Road vehicles — Functional safety*, 2018.

[34] I. O. f. Standardization, *ISO/PAS 21448:2019, Road vehicles — Safety of the intended functionality*, 2019.

[35] D. Bacciu, A. Carta, D. Di Sarli, C. Gallicchio and S. Petroni, "Towards Functional Safety Compliance of RecurrentNeural Networks," *under review*, 2021.

[36] A. Kurakin; I. Goodfellow; S. Bengio; et al., "Adversarial examples in the physical world," 2016.

[37] C.-Y. Ko, Z. Lyu, L. Weng, L. Daniel, N. Wong and D. Lin, "POPQORN: Quantifying Robustness of Recurrent Neural Networks," in *ICML*, 2019.

[38] P. Koopman, *Safety Performance Indicators (SPIs) for Self-Driving Cars,* 2020.

[39] R. Al-amri, R. K. Murugesan, M. Man, . A. F. Abdulateef, M. A. Al-Sharafi and A. A. Alkahtani, "A Review of Machine Learning and Deep Learning Techniques for Anomaly Detection in IoT Data," *Applied Sciences,* vol. 11, no. 12, p. 5320, 2021.

[40] G. Pang, C. Shen, L. Cao and A. van den Hengel , "Deep Learning for Anomaly Detection: A Review," *arXiv e-prints,* p. arXiv:2007.02500, 2020.

[41] L. Basora, X. Olive and T. Dubot, "Recent Advances in Anomaly Detection Methods Applied to Aviation," *Aerospace,* vol. 6, no. 11, 2019.

[42] R. Chalapathy and S. Chawla, "Deep Learning for Anomaly Detection: A Survey," *arXiv e-prints,* p. arXiv:1901.03407, 2019.

[43] P. Malhotra, L. Vig, G. Shroff and P. Agarwal, "Long Short Term Memory Networks for Anomaly Detection in Time Series," in *European Symposium on Artificial Neural Networks, Computational Intelligence,* Bruges, Belgium, 2015.

[44] P. Malhotra, A. Ramakrishnan, G. Anand, L. Vig, P. Agarwal and G. Shroff, "LSTM-based Encoder-Decoder for Multi-sensor Anomaly Detection," *arXiv e-prints,* p. arXiv:1607.00148, 2016.

[45] P. Schmidt, A. Reiss, R. Duerichen and K. V. Laerhoven, "Introducing WESAD, a Multimodal Dataset for Wearable Stress and Affect Detection," in *ICMI,* 2018.

[46] A. Cossu, D. Bacciu, A. Carta, C. Gallicchio and V. Lomonaco, "Continual Learning with Echo State Networks," in *European Symposium on Artificial Neural Networks (ESANN)*, 2021.

[47] Z. Li and D. Hoiem, "Learning without forgetting," *IEEE transactions on pattern analysis and machine intelligence,* 2017.

[48] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho and A. Grabska-Barwinska, "Overcoming catastrophic forgetting in neural networks," in *Proceedings of the national academy of sciences*, 2017.

[49] I. Goodfellow, Y. Bengio and A. Courville, Deep Learning, MIT Press, 2016.

[50] Y. LeCun, Y. Bengio and G. Hinton, "Deep learning," *Nature,* vol. 521, no. 7553, pp. 436-444, 2015.

[51] D. Bacciu, D. Di Sarli, C. Gallicchio, A. Micheli and N. Puccinelli, "Benchmarking Reservoir and Recurrent Neural Networks for Human State and Activity Recognition," *International Work-Conference on Artificial Neural Networks,* 2021.