SEMIoTICS

Deliverable D4.10
Embedded Intelligence and Local Analytics (final)

Deliverable release date	30.04.2020
Authors	1. Danilo Pau, Mirko Falchetto (ST) 2. Arne Broering (SAG) 3. Philip Wright (ENG) 4. David Parra (UP) 5. Lukasz Ciechomski, Pawel Gawron (BS) 6. Christos Tzagkarakis, Manolis Michalodimitrakis (FORTH)
Responsible person	Danilo Pau, Mirko Falchetto (ST)
Reviewed by	Konstantinos Fysarakis (STS), Christos Tzagkarakis, Nikolaos Petroulakis (FORTH), Jordi Serra (CTTC)
Approved by	PTC Members (Vivek Kulkarni, Nikolaos Petroulakis, Ermin Sakic, Mirko Falchetto, Domenico Presenza, Christos Verikoukis) PCC Members (Vivek Kulkarni, Nikolaos Petroulakis, Christos Verikoukis, Georgios Spanoudakis, Domenico Presenza, Danilo Pau, Joachim Posegga, Darek Dober, Kostas Ramantas, Ulrich Hansen)
Status of the Document	Final
Version	1.0
Dissemination level	Public

Table of Contents

ACRONYMS TABLE

Acronym	Definition
AI	Artificial Intelligence
ANN	Artificial Neural Network
ARIMA	Autoregressive Integrated Moving Average
ASC	Audio Scene Classification
AWS	Amazon Web Services
BLE	Bluetooth Low Energy
CDT	Change Detection Test
CNN	Convolutional Neural Network
CPM	Change Point Method
CPU	Central Processing Unit
DCNN	Deep Convolutional Neural Network
DL	Deep Learning
DR	Dynamic Reservoir
DSP	Digital Signal Processor
DTC	Deep Temporal Clustering
ESN	Echo State Network
FL	Federated Learning
FW	Firmware
GPU	Graphics Processing Unit
GRNN	Generalized Regression Neural Network
HAR	Human Activity Recognition
HW	HardWare
IHES	Intelligent Heterogeneous Embedded Sensors
IMU	Inertial Measurement Unit
IIoT	Industrial Internet of Things
IoT	Internet of Things
KPIs	Key Performance Indicator(s)

KWS	KeyWord Spotting
LA	Local Analytics
LIDAR	Light Detection And Ranging
LSM	Liquid State Machine
LSTM	Long Short Term Memory
mW	milliWatts
mJ	milliJoules
MCU	Micro Controller Unit
MHz	Mega Hertz (unit of frequency)
ML	Machine Learning
NPU	Neural Network Processing Unit
PERT (chart)	Program Evaluation Review Technique (chart)
RAM	Random Access Memory
RBML	Rule Based Machine Learning
RGB-D	Red Green Blue Depth (images)
RNN	Recurrent Neural Network
ROM	Read Only Memory
SARA	Socially Assistive Robotic Solution for Ambient assisted living
SAS	Self-Adaptive System
SoC	System on Chip
SOTA	State Of arT Analysis
SVM	Support Vector Machine
SW	SoftWare
TF	TensorFlow (DL Tool)
UCx	Use Case x (where x = 1, 2, 3)

1. INTRODUCTION

This deliverable is the final output of Task 4.3 (“Embedded Intelligence and local analytics”) and, as such, it focuses on the final definition, description and implementation of the functional blocks required for deploying (local) embedded intelligence at the IIoT/IoT Field Device Level in SEMIoTICS framework. More specifically, it aims to define Local Analytics (LA) embedded mechanisms in the project final architecture, to enable semi-autonomic local reaction/adaptation within IIoT/IoT devices. Task 4.3 and thus this deliverable exploits part of the work done on Task 4.2, where a preliminary analysis of the concepts related to analytics has been identified and defined. Moreover, in Task 4.2 this research has been exploited and finalized as part of D4.9 – “Monitoring, prediction and diagnosis mechanisms (final)”, to identify promising analytics methodologies deployable at all levels of the architecture, addressing the monitoring specific aspects, aiming at identifying the most relevant technology enablers to be used for defining the monitoring component of the architecture. On the other hand, the initial research done in the 1st draft of the deliverable (i.e., in D4.3 – “Embedded Intelligence and Local Analytics (first draft)”) has identified a subset of those algorithms, addressing the task of time series processing from sensing data, suitable for a real *lightweight* deployment at the Field Device level on computationally-constrained devices. The content of this deliverable has been adapted and updated from former D4.3, to provide a full characterization of SEMIoTICS selected algorithms among the ones initially mentioned. Also new algorithms have been introduced to support in D4.10 to better support all need of SEMIoTICS considered scenarios.

An overview of them is thus provided in section 2 and 3 of the deliverable with the intent to have a complete overview of today’s technical landscape to identify the subset of algorithms and tools that will be mapped to the SEMIoTICS framework as part of the integration activities in WP5.

In this context and considering the delta to the previous version of the deliverable, i.e. the D4.3, the latest developments and achievements presented within this final task 4.3 deliverable includes:

- All sections have been reviewed reflecting all the activities done in the reference period.
- An overview of the requirements identified in D2.3 used for the development of the algorithms has been added in section 0.
- All the algorithms have been implemented to support declared scenarios (as declared in D2.2), and where appropriate, the testing methodology and results have been presented as well.
- A new section 0 detailing a the set of all developed algorithms has been added to report the final set of identified tools, methodologies and algorithms, leveraging both Statistical and Artificial Intelligence (AI) methods in SEMIoTICS as part of the Local Embedded Analytics Component (section 3).
- Former D4.3 section 3.4 has been moved to section 4.1 in order to centralize in a single paragraph the presentation of all algorithms developed in SEMIoTICS.
- About the Gait Analysis problem in UC2 scenario, we added two more sections, one, starting from the generic introduction provided in section 2.3.4, we described in section 4.3 the AI algorithm in detail, focusing on some part of the code used. In particular, using Tensorflow and Keras DL Tools, written in Python language, it is described the list of layer instances passed to the constructors, the encoder and the decoder, and we focused on the implementation of these two models. We also present the verification between the two constructors described and the first draft for the implementation of the clustering on the Cloud, that it will be implemented as the next step for the development, together with the network training.
- A new approach based on Federated Learning has been added to the deliverable motivated by the UC1 scenario (see point above).
- Three new algorithms, not originally described in D4.3, have been added, detailed and characterized:
  - Oil Leakage Detector - based on Federated Learning approach (UC1)
  - Change Detection Point – CDT (UC3)
  - Change Point Method – CPM (UC3)

To present all the above, the deliverable is organized as follows:

- Section 1 is the present introduction explaining the structure and scope of the present deliverable, the SEMIoTICS PERT chart and a quick review of the requirements considered during Task 4.3 activities.
- Section 1.3 introduces and provides an overview of the envisaged approach in SEMIoTICS for what concerns the “Local Embedded Analytics”, with a short definition and a declaration of the main motivations and technical analysis that allows defining this key component of the project at the consortium level.
- Section 3 focuses more specifically on Artificial Intelligence (AI) and Machine Learning (ML) algorithms. ML algorithms are a subset of all local analytics algorithms adopted within SEMIoTICS. Differently from hand-crafted approaches, AI/ML algorithms require specific development policies and supporting tools. Thus, this section introduces in more detail the intended scenario together with a preliminary introduction of a typical ML algorithm development and deployment workflow using Deep Learning (DL) tools, from a perspective of the specific challenges of the deployment at Field Device level. Regarding the DL tools, an exhaustive overview is provided in this deliverable: section 3.3 elaborates on the DL tools adopted during the development of the local embedded analytics in SEMIoTICS. A short insight about envisaged deployment as part of use cases specific deployment (part of WP5 tasks) is provided as well, however, this aspect will be extensively covered in WP5 deliverables.
- Section 0 is a newly added section, not present in previous D4.3, that has been developed from former section 3.4 to properly characterize the algorithms that has been developed in task 4.3. It reflects the majority of efforts done in last Task 4.3 period during D4.10 activities for addressing the local analytics at field level in all three SEMIoTICS use case scenarios (UC1, UC2 and UC3).
- Section 4.7 derives the conclusions of the deliverable, the final implementation status that concludes WP4 related activities and the next planned steps within WP5 integration activities.
- Section 6 contains the bibliography and references mentioned on this deliverable.

1.1. PERT chart of SEMIoTICS

This section quickly introduces an overview on a per-task / WP basis of the interactions between tasks in SEMIoTICS, with specifically T4.3 depicted in yellow. The PERT chart is kept on task level for better readability.

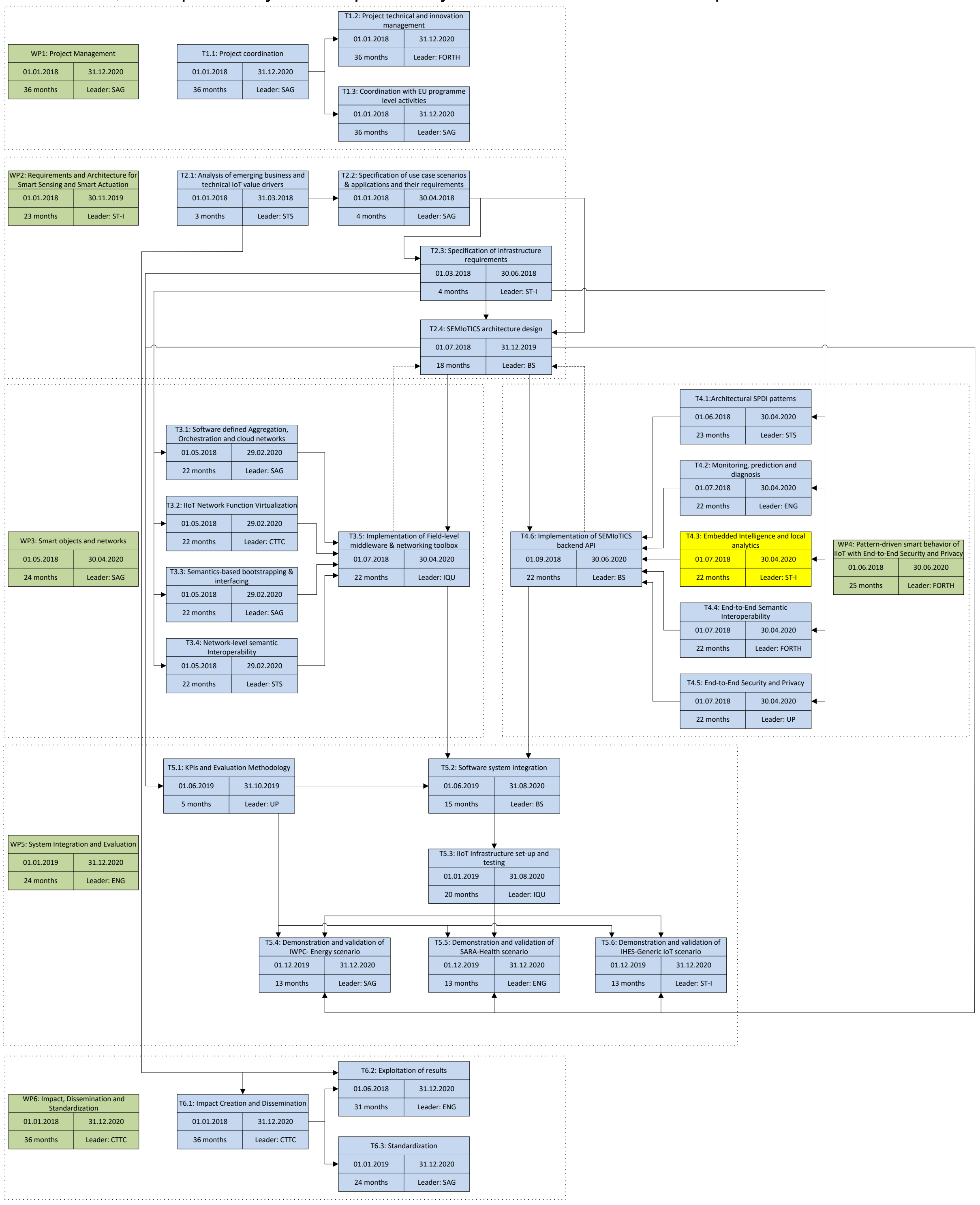

WP1: Project Management
01.01.2018 31.12.2020
36 months Leader: SAG
T1.1: Project coordination
01.01.2018 31.12.2020
36 months Leader: SAG
T1.2: Project technical and innovation management
01.01.2018 31.12.2020
36 months Leader: FORTH
T1.3: Coordination with EU programme level activities
01.01.2018 31.12.2020
36 months Leader: SAG
WP2: Requirements and Architecture for Smart Sensing and Smart Actuation
01.01.2018 30.11.2019
23 months Leader: ST-I
T2.1: Analysis of emerging business and technical IoT value drivers
01.01.2018 31.03.2018
3 months Leader: STS
T2.2: Specification of use case scenarios & applications and their requirements
01.01.2018 30.04.2018
4 months Leader: SAG
T2.3: Specification of infrastructure requirements
01.03.2018 30.06.2018
4 months Leader: ST-I
T2.4: SEMIoTICS architecture design
01.07.2018 31.12.2019
18 months Leader: BS
T3.1: Software defined Aggregation, Orchestration and cloud networks
01.05.2018 29.02.2020
22 months Leader: SAG
T3.2: IIoT Network Function Virtualization
01.05.2018 29.02.2020
22 months Leader: CTTC
WP3: Smart objects and networks
01.05.2018 30.04.2020
24 months Leader: SAG
T3.3: Semantics-based bootstrapping & interfacing
01.05.2018 29.02.2020
22 months Leader: SAG
T3.4: Network-level semantic Interoperability
01.05.2018 29.02.2020
22 months Leader: STS
T3.5: Implementation of Field-level middleware & networking toolbox
01.07.2018 30.04.2020
22 months Leader: IQU
T4.6: Implementation of SEMIoTICS backend API
01.09.2018 30.06.2020
22 months Leader: BS
T4.1:Architectural SPDI patterns
01.06.2018 30.04.2020
23 months Leader: STS
T4.2: Monitoring, prediction and diagnosis
01.07.2018 30.04.2020
22 months Leader: ENG
T4.3: Embedded Intelligence and local analytics
01.07.2018 30.04.2020
22 months Leader: ST-I
T4.4: End-to-End Semantic Interoperability
01.07.2018 30.04.2020
22 months Leader: FORTH
T4.5: End-to-End Security and Privacy
01.07.2018 30.04.2020
22 months Leader: UP
WP4: Pattern-driven smart behavior of IIoT with End-to-End Security and Privacy
01.06.2018 30.06.2020
25 months Leader: FORTH
T5.1: KPIs and Evaluation Methodology
01.06.2019 31.10.2019
5 months Leader: UP
T5.2: Software system integration
01.06.2019 31.08.2020
15 months Leader: BS
WP5: System Integration and Evaluation
01.01.2019 31.12.2020
24 months Leader: ENG
T5.3: IIoT Infrastructure set-up and testing
01.01.2019 31.08.2020
20 months Leader: IQU
T5.4: Demonstration and validation of IWPC- Energy scenario
01.12.2019 31.12.2020
13 months Leader: SAG
T5.5: Demonstration and validation of SARA-Health scenario
01.12.2019 31.12.2020
13 months Leader: ENG
T5.6: Demonstration and validation of IHES-Generic IoT scenario
01.12.2019 31.12.2020
13 months Leader: ST-I
T6.2: Exploitation of results
01.06.2018 31.12.2020
31 months Leader: ENG
WP6: Impact, Dissemination and Standardization
01.01.2018 31.12.2020
36 months Leader: CTTC
T6.1: Impact Creation and Dissemination
01.01.2018 31.12.2020
36 months Leader: CTTC
T6.3: Standardization
01.01.2019 31.12.2020
24 months Leader: SAG

FIGURE 1: SEMIOTICS PERT CHART (PER-TASK)

1.2. Task 4.3 related Requirements

A major contribution in consolidating this final version of deliverable D4.10, in order to address the task of developing the local embedded analytics at Field Device layer in SEMIoTICS, is provided by the three major use cases identified in D2.2 within the project and their derived requirements described in the D2.3. In particular, in relation to task 4.3 development phase the following requirements subset has been considered. Here is a short summary from D2.3 considered during task 4.3 activities impacting the local analytics:

- D2.3-Table1-R.GP.1: End-to-end connectivity between the heterogeneous IoT devices (at the field level) and the heterogeneous IoT Platforms (at the backend cloud level)
- D2.3-Table1-R.GP.2: Scalable infrastructure due to the fast-paced growth of IoT devices
- D2.3-Table4-R.FD.1-15: some subset of these generic field devices requirements were considered for developing the local embedded analytics part, such as the capability of acquiring and processing data in real-time from environment, events aggregation and clustering, bandwidth minimization, interaction with IoT gateway, interoperability via standard protocols (e.g. JSON) and communication infrastructure (e.g. Rest APIs, CoAP, MQTT, etc.)
- D2.3-Table7-R.UC1.10: Local analytical capability of IIoT Gateway to run machine learning algorithms
- D2.3-Table8-R.UC2.15: The Robotic Rollator (RR) exploiting AI Services to analyze Patient’s gait and posture to identify significant events
- D2.3-Table8-R.UC2.17: The SEMIoTICS connectivity SHOULD support real time exchange of raw sensor data
- D2.3-Table8-R.UC3.1: IoT Sensing unit shall be able to embed environmental and inertial sensors
- D2.3-Table8-R.UC3.3: IoT Sensing unit shall be able to learn a model from observed data
- D2.3-Table8-R.UC3.4: IoT Sensing unit shall be able to detect relevant changes from the learned model
- D2.3-Table8-R.UC3.5: IoT Sensing unit shall be able to adapt to a new model
- D2.3-Table8-R.UC3.6: IoT Sensing gateway shall be able to coordinate a set of IIoT sensing units
- D2.3-Table8-R.UC3.11: IoT Sensing unit shall be able to run Artificial neural networks on the MCU in real time
- D2.3-Table8-R.UC3.12: IoT Sensing unit shall be able to run lightweight statistical model analysis algorithms
- D2.3-Table8-R.UC3.14: MCU IoT Sensing unit shall be able to run neural network online training at the sensor data rate of choice
- D2.3-Table8-R.UC3.17: IoT Sensing gateway shall be able to negotiate capabilities, notify, start and shutdown any Sensing unit
- D2.3-Table8-R.UC3.24: Sensing units may be equipped with dedicated FW to detect relevant sensors malfunctioning and report that to the gateway

In particular, for the UC2 scenario the generic requirements GP has been taken into account for developing the gait analysis scenario.
About the UC3 scenario, the key focus of the task 4.3 activities was to enable local analytics on ST Microelectronics STM32 MCU devices, equipped with embedded sensors (inertial + environmental ones). These sensor raw data values are processed using statistical methods and AI/ML algorithms. This approach aims to minimize the transmission bandwidth of raw data in favor of a smarter communication triggered by relevant events coming from the on-board signal processing. This must be based on robust and well-known communication protocol that allow the association of new devices to the network and the communication between the analytics devices and the IIoT/IoT Gateway. For a better understanding and a detailed explanation of all the requirements, consider reading section 2 and 3.3 of D2.3
In particular, these mechanisms will be implemented close to the sensing hardware (i.e., a set of smart MCUs equipped with communication capabilities), taking advantage of the ability to autonomously detect changes in the real-time acquired data streams, e.g., induced by faults affecting sensors and actuators or sensed in a time-

variant environment. The majority of the *Local Embedded Algorithms* adopted in SEMIoTICS will be derived from a novel signal processing design methodology presented in section 2.3.3, together with an embedded resource-constrained technological implementation affecting the sensor acquisitions in groups of heterogeneous sensing nodes.
This methodology requires:

1. Online learning of the signal model using a minimal number of samples (D2.3-Table8-R.UC3.3).
2. Apply the learnt model to compute a residual signal using input samples and predictions (D2.3-Table8-R.UC3.3).
3. Design a model-free change detection test applied to residual signal (D2.3-Table8-R.UC3.4).
4. Design a change-point method to validate the detected change (D2.3-Table8-R.UC3.4).

1.3. Task 4.3 related KPIs

A subset of the KPIs reported in D5.1 – "SEMIoTICS KPIs and Evaluation Methodology" are related also to the efficient deployment and mapping of local embedded analytics algorithms within the SEMIoTICS framework. The technical choices that are presented in this final version of the deliverable were not only driven by the requirements specified in D2.3, but they were made also in consideration of some KPI figure of interest. An example is the algorithmic complexity described in section 0, including the deployment constraints, and the power consumption analysis anticipated in some of the algorithms presented (e.g. see section 4.4.2.2).
A complete presentation of KPIs reported in Table 1, related to local embedded analytics impact in SEMIoTICS project, is deferred to WP5 activities. In fact, the majority of these KPIs are related to characterize and measure relevant performance figures. Thereby, these assessments will be done as part of tasks from task 5.4 to task 5.6 of WP5, devoted to the deployment, demonstration and validation of all SEMIoTICS use case scenarios.

TABLE 1. TASK 4.3 RELEVANT KPIS

SPDI Patterns	KPI ID (reported in T5.1)	KPI Description	Status
Multi-layered Embedded Intelligence	KPI-4.1	Delivery of lightweight ML algorithms	**DONE:** a total of six ML/Statistical Algorithms has been developed in T4.3 and reported into Section 4 of D4.10.
	KPI-4.2	Delivery of mechanisms with adaptation time of 15ms	Deferred to WP5 validation and testing of the FL monitoring infrastructure.
	KPI-4.3	Delivery of adaptations mechanisms enabling improvement by at least 20%	Deferred to UCs demo deployment validation and testing.
	KPI-4.4	Detection time of less than 10 ms	Deferred to T5.6 IHES final testing. Preliminary benchmarks done in task 4.3 shows CDT takes <=10ms time for each input.
	KPI-4.5	Baseline improvement of 20% adaptation time	Deferred to T5.4 to T5.6 as part of UCs demo deployment.
	KPI-4.6	Development of new security mechanisms/controls	Deferred to T5.5 and T5.6 testing.

2. EMBEDDED INTELLIGENCE IN IOT/IIOT ENVIRONMENTS

2.1. Embedded Intelligence in SEMIoTICS

A key goal of the SEMIoTICS project is to provide a reference infrastructure for supporting multi-layered *Embedded Intelligence*. This means that at each layer of the SEMIoTICS architecture there are specific components (see D2.5 – "SEMIoTICS Architecture (final)" for further details) where embedded intelligence algorithms will be deployed. These components will export coherent APIs, specific at each logical level of the architecture, to make those algorithms available to other components of the architecture. Several algorithms will be implemented during the project's lifespan; embedded intelligence will be demonstrated by covering different implementation scenarios exploiting the use cases defined in D2.2 – *SEMIoTICS usage scenarios and requirements* – in compliance with the requirements defined in D2.3. In Figure 2, the generic approach envisaged within the SEMIoTICS framework is visualized. In this context, this deliverable focuses on the specific task of deploying Embedded Intelligence at the IoT/IIoT Field Devices level. Part of the analytics processing is mapped also at IoT/IIoT Gateway in SEMIoTICS, mainly for supporting monitoring and prediction diagnosis on the SEMIoTICS Field Devices. Please refer to D4.9 for a complete presentation about these implementation details at IoT/IIoT Gateway level.

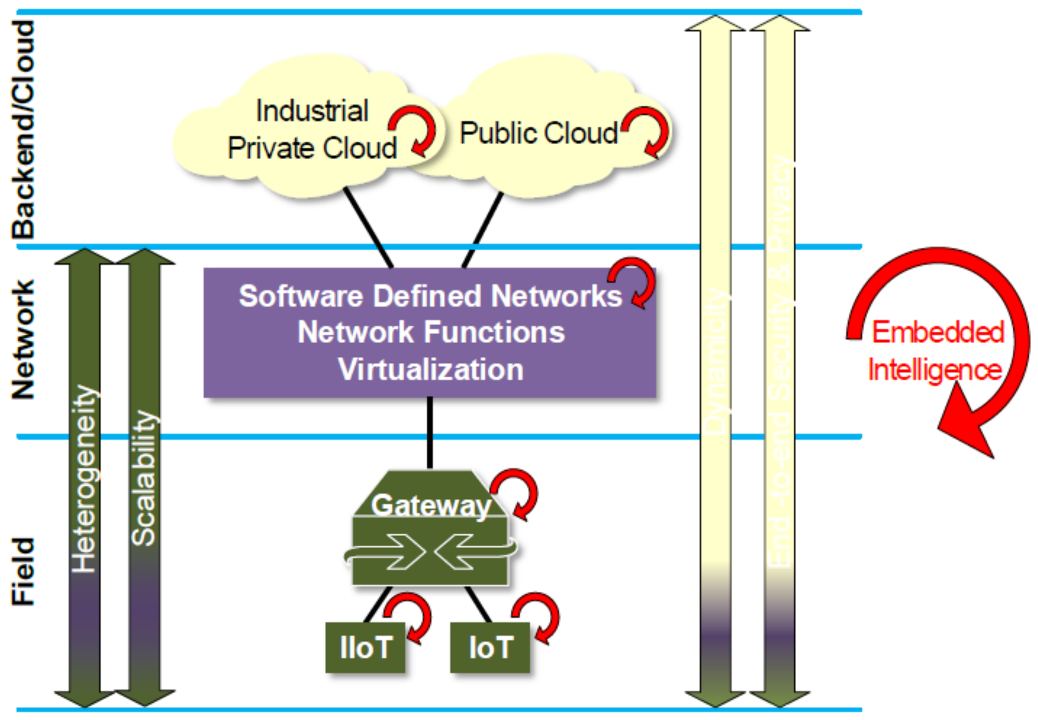


FIGURE 2: MULTI-LAYERED INTELLIGENCE IN SEMIOTICS

In SEMIoTICS "embedded intelligence" corresponds to the generic aspects dealing with the definition of the features/infrastructure/algorithms set/supporting tools implementing the concepts referring to the local analytics (see next section for further details). In this deliverable, we will cover these aspects focusing on the challenges and implications of porting such kinds of functionalities at the IIoT/IoT Field device level. We will use the term ***"Local Embedded Intelligence"*** and "***Local Analytics***" to refer to this level of SEMIoTICS architecture.

2.2. IIoT/IoT Field Devices Local Analytics

There are 150 billion embedded processors in the world, which is more than twenty for everyone on Earth and the growth rate is 20% annually, with no signs of slowdown (see Figure 3). The fundamental concern of the Internet of Things (IoT) is that the majority of the actual embedded devices are not really connected to any network and it is unlikely that they ever will be, at least more than intermittently. This sounds like a paradox, but the majority of the devices do not need either persistent connection to the network, because it is not strictly required, or not feasible at all for many batteries powered tasks. What these devices really need is to become smarter and able to locally process incoming data to extract relevant features from them. A good example are the wearable devices used for health monitoring or fitness activities. The main constraint embedded devices face is energy and communication. Wiring them into power units or connecting them with a reliable wireless connection is hard or impossible in most environments. The maintenance burden of replacing batteries quickly becomes unmanageable as the number of devices increases. The only way for the number of devices to keep increasing is if they have batteries that last a very long time, or if they can use energy harvesting (e.g., solar cells from indoor lighting).

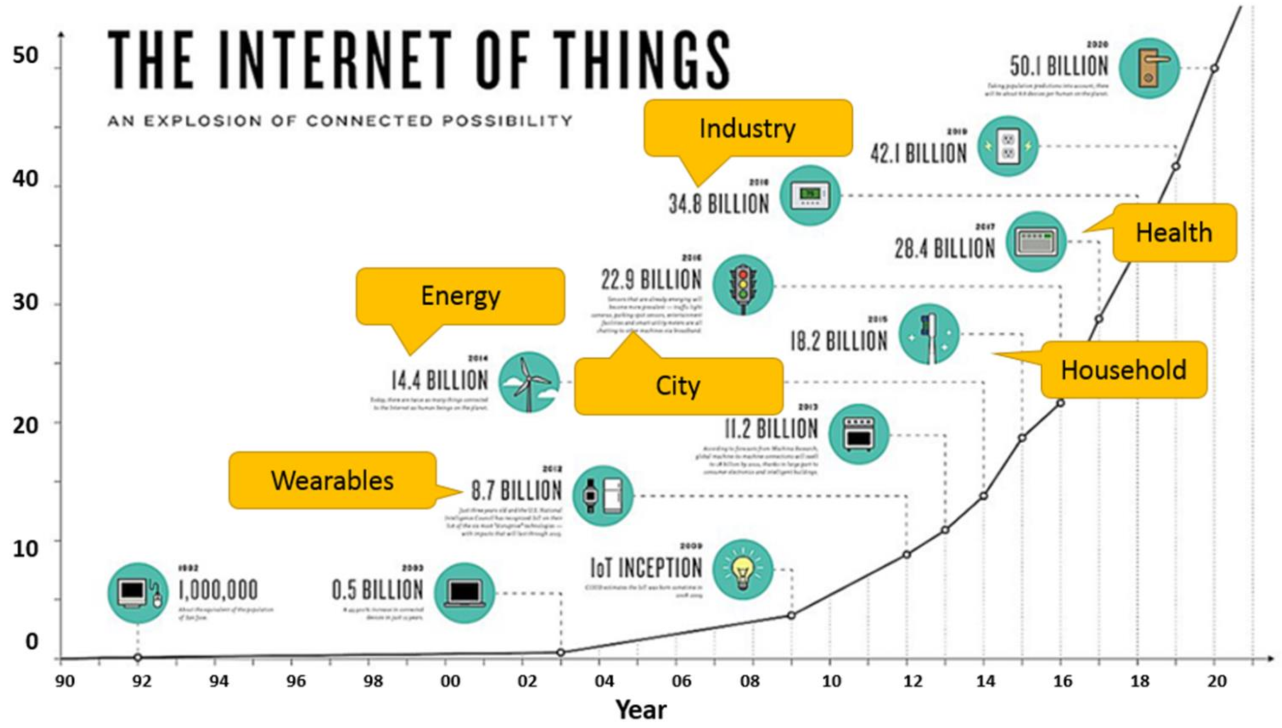


FIGURE 3: IIOT/IOT CONNECTED DEVICES GROWTH[1]

Constraining energy aims to keep energy usage at the milliWatts level or even lower. This is essential to give a long time of continuous use on a reasonably small and cheap battery, or alternatively, it is within the range of a decent energy harvesting system like ambient solar. In addition, anything involving radio takes a lot of energy, far more than one milliWatts in most cases. Even when low power radio devices are available[1], they are so simple designed that it is not possible to do any kind of local processing. Transmitting bits of information, even with protocols like Bluetooth Low Energy (BLE), a wireless point-to-point protocol specifically designed for low power devices, is in the tens to hundreds of milliWatts in the best-case scenarios and comparatively short range (typically 10 meters). The efficiency of radio transmission does not seem to be improving dramatically over time either; there seem to be some tough obstacles imposed by physics that make improvements hard. If a system uses Wi-Fi the power consumption challenge is just dramatically exacerbated.

[1] Courtesy from ST slide on IoT Device growth trends. Adapted from original ncta figures.

Capturing data through sensors and process them locally does not suffer from the same problem. There are microphones, accelerometers and even image sensors that operate well below a milliWatts, even down to tens of microwatts. The same is true for general Microprocessors and Digital Signal Processors (DSPs) that can process tens or hundreds of millions of calculations for under a milliWatts, even with existing technologies, and much more efficient low-energy accelerators are on the horizon. Therefore, most data captured by sensors in the embedded world is just being discarded, without being analyzed at all. This is the point where SEMIoTICS, developing the concept of multi-layered intelligence, aims at bringing a novel approach and skills to analyze such data in an intelligent supervised[2]/unsupervised[3] new way that fits into embedded low power micro controllers. The key rationale supporting the multi-layered intelligence is the simple observation that the bottleneck of a massive scalable IIoT/IoT distributed system is not on the local processing of sensed data, but actually on the transmission of those data to the upper levels of the architecture. As long as the connected devices number will grow, current mainstream approaches to (remote) data processing will not be sustainable in terms of infrastructure and power needs. SEMIoTICS will demonstrate the effectiveness of this new local analytics driven approach in a relevant use case scenario as part of WP5 activities. These local analytics will also support and complement another key aspect of SEMIoTICS that is the pattern driven approach: this approach means (also) to avoid, e.g., to propagate un-necessary raw data from sensors and optionally to propagate some relevant event generated as result of local data-reduction algorithms instead. This flow is coherent with the pattern approach followed in SEMIoTICS and is represented briefly in Figure 4 below. This figure overviews a typical Local Analytics data/event flow processing where at IIoT/IoT Field Devices data are processed to produce and transmit relevant events (saving power) to IIoT/IoT Gateway. These events are then further elaborated at that level (e.g., by exploiting data correlations to analyze data dependencies between nodes). Finally, further events (e.g., the data dependencies graph) may be sent to the Backend/Cloud level where further data aggregation, clustering, analytics could be performed. This is shortly a simplified example of *multi-layered intelligence*.

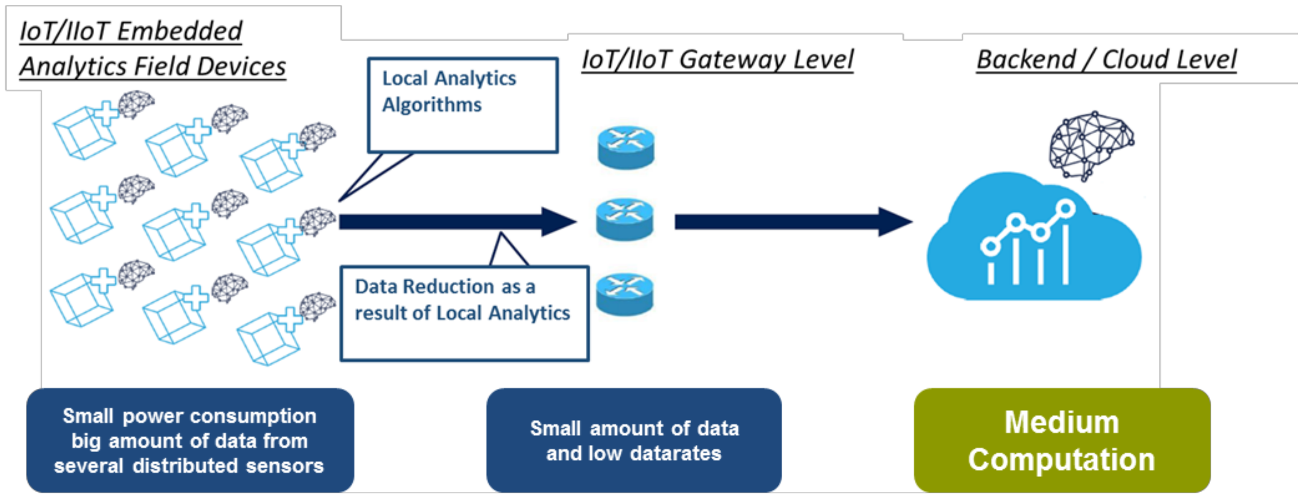

IoT/IIoT Embedded Analytics Field Devices
Local Analytics Algorithms
IoT/IIoT Gateway Level
Backend / Cloud Level
Data Reduction as a result of Local Analytics
Small power consumption big amount of data from several distributed sensors
Small amount of data and low datarates
Medium Computation

FIGURE 4: SEMIOTICS LOCAL ANALYTICS FLOW

In SEMIoTICS, Local Analytics (LA) algorithms will be in general mapped as specific components with a clear API interface to exploit their functionalities and services. Therefore, this deliverable D4.10 focuses on local embedded intelligence. Thus, the LA algorithms considered here will be directly mapped as close as possible to the data source (i.e., the sensor). As a result, we assume that the data processing takes place directly on MCU powered Field Devices or at the local IIoT/IoT Gateway. Specific characteristics and features of the deployment platform must be taken into consideration, where we consider specific legacy HW/SW as well: coherently, multiple dedicated components will be implemented and tested.

[2] https://en.wikipedia.org/wiki/Supervised_learning
[3] https://en.wikipedia.org/wiki/Unsupervised_learning

One of the envisaged scenarios is to demonstrate LA considering a real-time scheme, and to focus its use on the processing of data streams, where:

- Typical inputs are data from sensors. In SEMIoTICS, we consider data streams provided by slow time varying environmental sensors (i.e., temperature/pressure/humidity/lighting) or high-varying inertial sensors (i.e., accelerometer, gyroscope, etc.).
- Typical outputs are data that depend on the adopted algorithmic procedures. For example, clustering algorithms usually compute on a given cluster set the probability that a given input belongs to the n-th cluster. Predictive AI/ML models instead predict the input at time N by observing e.g., W past samples acquired at times [N-W…, N-1]. Generalizing, under a wide range of conditions, it could be observed that the outputs may be seen as a sort of derived time series compacting input data. ML/LA algorithms could be seen as virtual sensors that generate data (at lower rates vs physical sensors), and thus can be considered as equivalent to any other type of dummy raw data sensor within the SEMIoTICS architecture.
- In some cases, the deployment of a physical component depends, as previously stated, on the boundaries imposed by the specific platform and on legacy middleware dependencies.
- For sensor data stream processing, the SEMIoTICS approach is the one inspired by data processing/reduction techniques where sensors (attached to field devices) are not directly connected to the backend layer, and never stream raw readings to that layer. Instead, they are locally connected to other intra-layer components on the same level. Interoperability is ensured by proper definitions of northbound/southbound inter-layer interfaces and routed through a controlled pattern driven design.

2.3. IIoT/IoT Field Devices Semi-Automatic Local Adaptation

One major objective in SEMIoTICS is to enable the concepts of multi-layered distributed intelligence. The main reasons motivating this property have been already discussed in the previous section. Here, we further analyze the details on how SEMIoTICS will accomplish this multi-layered intelligence concept. Multi-layered intelligence means that at all levels of SEMIoTICS, there are specific actors/components that implement some sort of smart analytics to achieve a particular task. Thus, in SEMIoTICS, the embedded analytics is functional to implement the concept of multi-layered intelligence. In general, embedded analytics will generally implement some sort of smart data processing and data reduction mechanism to make the architecture more scalable and reliable. In this respect, we envisage the need to use different types of algorithms and methods to implement a generic approach for handling the multi-layered intelligence. We identified three different families of algorithms that will be shortly described in the following subsections and they will be detailed more in the next deliverable cycle.

The type of algorithms under evaluation within the SEMIoTICS framework can be divided into two categories: statistical methods and Machine Learning (ML)/Artificial Intelligence (AI) methods. There is the misleading perception and confidence that AI algorithms could often (if not always) solve any kind of problem, no matter what the intended use is for: processing, classification, regression, prediction, etc. This is not the case, since AI, like any other technology field, has both its own pros and cons: depending on the specific application and requirements, an AI algorithm could be an oversized solution when a simpler (and faster) solution can be implemented using valid alternatives. Moreover, DL and ML methods are not applicable everywhere; a few of drawbacks that these algorithms have are shortly mentioned below:

- **Inference latency:** In some applications the latency requirements for a single inference are close or below millisecond time: depending on the target platform, not many convolutions are feasible in this short time. In those cases, simpler linear or autoregressive models offers a proper alternative (and often they are still a good solution for the problem).
- **Outcome reproducibility:** Deep learning algorithms sometimes have a stochastic behavior: putting them to work in real-life, the same DL model trained twice on the same data may converge to the same overall loss and quality metrics, but behave differently on real input data. In some

systems, e.g. in automotive or avionics domains, unpredictable behavior of your model because of a retraining is not recommended.

- **Learning complexity:** The more coefficients are required to train, the more (labelled) data are needed for training a good model: a big neural network generally needs more training samples to converge than a simple linear model. A related problem is also the dimensionality of the data: more and more features need to be represented in the model and usually this impact on the sparsity of the data representation, since in general a higher dimensional space tends to become sparse as well. The final consequence is that the number of parameters to be trained heavily impacts the needed training data size (usually an exponential growth).
- **(Good) Legacy algorithms:** There is already a number of non-deep learning models that are available, already tested and fully working. Even if the trend today is to apply deep learning everywhere, rethinking AI powered solutions for well solved problems can end-up in a lot of resources and time waste.

2.3.1. STATISTICAL METHODS

Statistical methods try to analyze relations from observed data. Observed data in the scope of this deliverable are focused on IIoT/IoT device lightweight algorithms, with a generic time variant signal that comes from a sensing node. Having a set of sensed data acquired from a set of sensors, an interesting objective could be to estimate a set of statements (i.e., relations) regarding the data between the measured variables in order to correlate them: usually, a generic graph representation is used as typical outputs of these kind of algorithms. In SEMIoTICS, we will try to shape those algorithms by evaluating them in real world conditions, thus on realistic data.

A fundamental concept within the embedded intelligence framework is the use of statistical based methods that are used for prediction. In the case of time series, where data is gathered from various IIoT/IoT devices/sensors, the task of predicting the future behavior of the observed time series is very important, especially in cases such as predictive maintenance (e.g., predicting a machine temperature etc.). For that reason, statistical techniques such as autoregressive models can be applied in this direction. More specific, the Auto-Regressive Integrated Moving Average (ARIMA) model is the most basic method used to model time series data for prediction/forecasting, in such a way that:

- a pattern of growth/decline in the data is accounted for (auto-regressive part)
- the rate of change of the growth/decline in the data is accounted for (integrated part)
- noise between consecutive time points is accounted for (moving average part)

ARIMA models[4] are typically expressed like ARIMA(*p, d, q*) in the bibliography, with the three terms *p*, *d*, and *q* defined as follows:

- *p* denotes the number of preceding (lagging) time series values that have to be added/subtracted to the time series, so as to make better predictions based on local periods of growth/decline in the observed data. This captures the autoregressive nature of ARIMA
- *d* corresponds to the number of times the data have to be differenced (by differencing the time series we can enforce stationarity to an initially non-stationary series) to produce a stationary series (i.e., a time series that has a constant mean over time). This captures the integrated nature of ARIMA. If *d=0*, this means that our data do not tend to go up/down in the long term (i.e., the model is already stationary). If *d* is 1, then it means that the data are going up/down linearly. If *d* is 2, then it means that the data are going up/down exponentially.
- *q* represents the number of preceding/lagging values for the error term that are added/subtracted to the time series. This captures the moving average part of ARIMA.

[4] https://otexts.com/fpp2/arima.html

Besides, within the statistical analysis framework, correlation can be used, which is any statistical association between two random variables. There are several metrics measuring the degree of correlation. The most common is the Pearson correlation coefficient, which is sensitive only to a linear relationship between two variables (which may be present even when one variable is a nonlinear function of the other). Mutual information can also be applied to measure dependence between two variables.
More details are provided in Section 4.2, where an experimental evaluation is given based on real environmental data.

2.3.2. MACHINE LEARNING (ML) – ARTIFICIAL INTELLIGENCE (AI) METHODS

One of the fundamental challenges of IIoT/IoT technology nowadays is the efficient and robust implementation of AI/ML algorithms in light of an embedded intelligence framework. Typically, AI/ML algorithms use large amount of training (offline labelled) data and virtually unlimited resources to perform the learning/prediction/classification etc. tasks. As a result, it is very important to focus on AI/ML algorithmic techniques that exploit the limited data directly collected via the IIoT/IoT devices/sensors without the need of transmitting further data or information to the upper IIoT/IoT infrastructure levels. In the current section, we provide a brief literature review about available algorithmic solutions that could be applied within the embedded intelligence framework. The SEMIoTICS platform can form a roadmap for our potential research work (as well as practical implementation work) on that subject.

More specifically, in [1] a tree-based algorithm is developed for efficient inference on IIoT/IoT devices having limited resources (e.g., 2KB RAM and 32KB read-only flash). The algorithm maintains prediction accuracy while minimizing model size and prediction costs by developing a tree model which learns a single, shallow, sparse tree with powerful nodes. Moreover, all data are sparsely projected into a low-dimensional space in which the tree is learnt and joint learning of all tree and projection parameters is performed. In [2] the authors introduce a compressed and accurate K-Nearest Neighbors algorithm for devices with limited storage.
The described approach is inspired by k-Nearest Neighbors but has several orders of magnitudes less storage and prediction complexity: a small number of prototypes is learnt to represent the entire training set, and then a sparse low dimensional projection of data is performed. Finally, joint discriminative learning of the projection and the prototypes with an explicit model size constraint is carried out.

Deep neural networks have been implemented to run on embedded devices by reducing redundancy in their parameters: in [3], the authors study techniques for reducing the number of free parameters in neural networks by exploiting the fact that the weights in learned networks tend to be structured. Additionally, neural network compression techniques such as quantization and encoding are presented in [4], where the authors introduce the "Deep Compression" scheme to compress the neural networks without affecting accuracy. The idea is that "Deep Compression" operates by pruning the unimportant connections, quantizing the network using weight sharing, and then applying Huffman coding.

In [5], the concept of "BinaryConnect" is proposed: it is a method consisting on a deep neural network training procedure with binary weights during the forward and backward passes, while retaining precision of the stored weights in which gradients are accumulated. Binary weights, i.e., weights which are constrained to only two possible values (e.g., -1 or 1), can bring great benefits to specialized deep learning hardware by replacing many multiply-accumulate operations with simple accumulations, as multipliers are the most space and power-hungry components of a digital implementation of neural networks. The authors in [6] describe the "HashedNets" architecture that exploits the inherent redundancy in neural networks to achieve a drastic reduction in model size. The proposed approach uses a low-cost hash function to randomly group connection weights into hash buckets, and all connections within the same hash bucket share a single parameter value. These parameters are tuned to adjust to the proposed neural network weight sharing architecture using the standard back-propagation process during training.

"LightNNs" framework is proposed in [7], which modifies the computation logic of conventional deep neural networks by making reasonable approximations, and replacing the multipliers with more energy-efficient operators involving only one shift or limited shift-and-add operations. In addition, the "LightNNs" approach also reduces weight storage, thereby decreasing the energy for memory accesses. In [8], the authors consider the task of building compact deep learning pipelines suitable for deployment on storage and power constrained mobile devices. They propose a unified framework to learn a broad family of structured parameter matrices that are characterized by the notion of low displacement rank. The proposed structured transforms admit fast function and gradient evaluation, and they span a rich range of parameter sharing configurations whose statistical modelling capacity can be explicitly tuned along a continuum from structured to unstructured. In [9] the authors describe the design, realization and full integration of a ML algorithm on a simple NXP sensor board typical of IIoT/IoT systems. The ML process relies upon on a Gaussian mixture model that uses the expectation-maximization algorithm with the minimum description length criterion. The implemented algorithm is based on a probabilistic model generated on the NXP board that characterizes the statistical features of sensor measurements in real time.

CMSIS-NN (see [10]) embodies efficiently Neural Network Kernels for ARM Cortex-M CPUs into a library. This is a software library with a collection of efficient neural network kernels developed by ARM to maximize the performance and minimize the memory footprint of neural networks on Cortex-M processor cores. The library is divided into a number of functions, each covering a specific category: Neural Network Convolution, Neural Network Activation, Fully-connected Layers, Neural Network Pooling, Softmax and Neural Network Support Functions. The library has separate functions for operating on different weight and activation data types including 8-bit integers (q7_t) and 16-bit integers (q15_t). The description of the kernels is included in the function documentation. The implementation details are also described in the published paper [10].

Tinn[5] (Tiny Neural Network) is a 200-line dependency-free neural network library written in C99 and uses no more than the C standard library. Tinn is meant for embedded systems. The concept is to train a model on a powerful desktop and load it on a microcontroller and use the analog-to-digital converter to predict real time events. Tinn can be multi-threaded.

e-AI[6] from Renesas is a development environment and an effective tool to embed Artificial Neural Networks (ANN) into an MCU/MPU after off-line learning. There are some difficulties in implementing a learned model on an MCU/MPU. Main reasons are: lack of Python language native support that is not compatible with optimal ROM/RAM management on MCU/MPU devices. So, there is a well-established trend where Python is used as a description language in many AI frameworks, while the control program of the MCU is usually written in C/C++. The e-AI development environment solves these problems and makes it possible to implement the learned ANN on an MCU/MPU in conformance with C/C++ projects.

Tensorflow[7] Lite is an experimental framework Google is sponsoring to derive a "tiny" framework specifically designed for micro controllers, which involves four major steps: create or find a (lightweight) model architecture, train a model off-line, convert the model, write code to run inference. This is a flow conceived by Google to achieve AI inference at the edge by exploiting the constrained resources of an MCU with few Kbytes of memory available. It is an interesting experiment, not yet mature nor widely adopted: it has still some limitation on managing optimal memory allocation and schedule for the mapped NN model, and in particular the last step still required hand-written code for actual mapping of the model on target MCU. In this respect the STM32Cube.AI tool presented in section 3.3.6 offers more advanced features, e.g. the automatic generation of the NN model code already integrated into ST CubeMX middleware software.

2.3.3. IIOT/IOT FIELD DEVICES UNSUPERVISED LOCAL ANALYTICS

[5] https://github.com/glouw/tinn
[6] https://www.renesas.com/eu/en/solutions/key-technology/e-ai/about.html
[7] https://www.tensorflow.org/lite/microcontrollers/get_started

One of the key objectives in SEMIoTICS is to support distributed intelligence at all layer of the architecture, following an edge-computing driven approach. Focusing on the field level, and more specifically to the field devices level, a major contribution is provided by the UC3 IHES demonstrator scenario. This embedded engineered system will be deployed within SEMIoTICS, mainly at the field device level by defining a set of lightweight algorithms mapped on heavily constrained STM32 MCUs devices (D2.3-Table8-R.UC3.3 requirement). These specific algorithms have been wrapped as a dedicated MCU software middleware (i.e., embedded binary firmware loaded into each MCU unit), in order to provide a fully working edge computing example in the UC3 scenarios.

The IHES system provides innovative algorithms and system libraries to enable real-time semi-automatic unsupervised local adaption, borrowing some concept from these research domains anticipated in this section and next section 3. A self-adaptive system (SAS) is a system able to adapt its behavior and / or structure in reference to changes in the system itself and/or its operating environment, in an autonomous way (D2.3-Table1-R.GP.2 requirement). These algorithms have been specifically designed, validated and engineered for real-time MCUs processing analytics.

All the algorithms described in these sections have been derived from the generic and UC3 specific requirements reported in D2.3 and shortly summarized in section 0 of this deliverable.

A specifically designed implementation, including linear predictive models and AI/ML nonlinear predictive algorithms (see section 4.1.3 and section 4.4 for further details) has been developed as part of task 4.3 activities. This implementation is supported by some specifically designed change detection algorithms with a chained validation phase to mitigate false positive changes has been developed and a full functional implementation has been developed in task 4.3, by mapping it to ST MCU STM32 devices as a dedicated C library firmware exposing dedicated interfaces and protocols to facilitate its integration as part of the UC3 demo scenario – T5.6 within WP5 activities.

This mapping demonstrates how it is possible to implement an efficient yet generic local data processing/analysis flow at a single MCU processor, keeping advanced functionalities yet providing way better energy consumption figures thanks to the low data rate communication flow. Moreover, some AI algorithm (see next section 3 for further details) will also be included as part of the running FW on the device to demonstrate that on this kind of tiny devices it is possible under specific assumptions to map a ML algorithm when it is carefully designed and implemented. The high detection accuracy together with the low computational load and memory occupation makes the proposed methodology (and its technological implementation) well suited for self-adaptive smart AI-powered sensing nodes employing low-cost high-volume micro controllers widely adopted in the mass market for the Internet of Things. A generic overview of the component functional architecture in a single MCU device is shown in Figure 5 below.

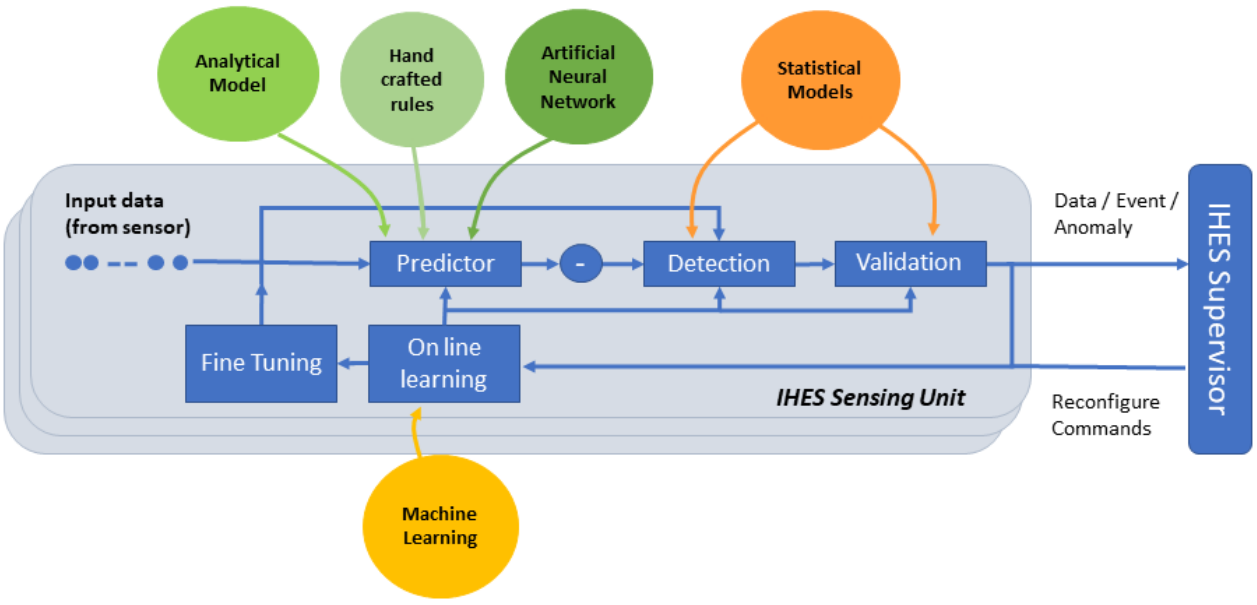


FIGURE 5: LOCAL LEARNING / ADAPTATION ALGORITHMS – IHES SENSING NODE

The IHES system represents the actual technological implementation of a novel methodology specifically defined for detecting changes affecting the sensor acquisitions in units of IIoT/IoT Intelligent Nodes. This methodology has been conceived to detect changes at the sensor level or close to them, increasing responsiveness, scalability and allowing the nodes to detect faults in the sensors or time-variance in the environment, without requiring any a-priori information nor assumption about the environment under inspection.

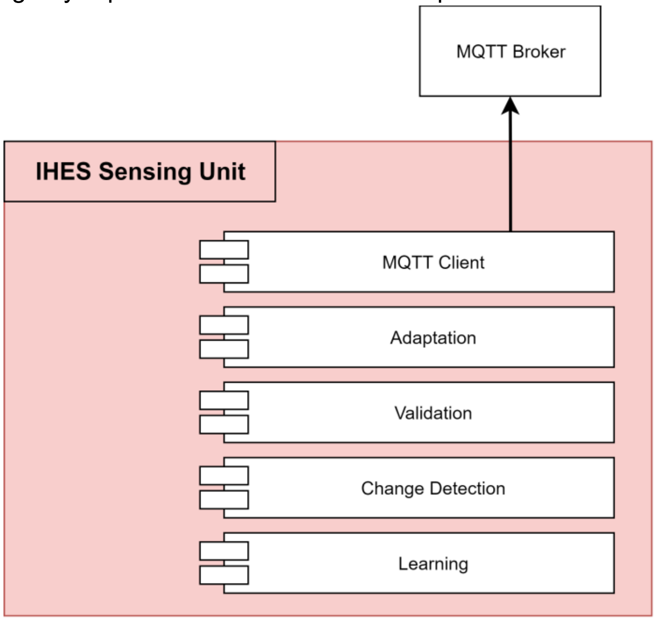


FIGURE 6: IHES SENSING NODE SW MIDDLEWARE

Considering the C written software middleware in Figure 6 aboveFigure 6, each IHES Sensing Unit is divided into five different modules which will be described in following sections with the aim of providing a comprehensive view of all algorithms that are altogether called *UC3 Local Embedded Analytics* as the set of

intelligent algorithms that operates directly in the sensor node for the purpose of transforming acquired raw data time variant signals into information (i.e., events) useful for higher levels of the architecture.
The novelty of the proposed approach applied to low-cost resource constrained micro controllers, both in terms of computational power and integrated RAM and ROM memories, resides in the definition of a processing pipeline implementing a state machine composed by initialization/learning and runtime modes operating over a shared global context on top of the developed SW modules. This state pipeline is mainly defined by:

- A learning mechanism to build the predictive model describing the sensor data stream over time.
  The online learning step relies on an initial change-free training sequence of samples acquired by the sensor. Once the learning phase over a relatively small sliding time-window has been completed, the discrepancy between the value measured by the sensor and the one estimated by the predictive model (the residual) running on micro controller is calculated. A fine-tuning stage has been added to the pipeline as part of the online training to avoid overfitting on the acquired model. This new feature will be detailed in section 4.4.1. After this learning phase has been done, the pipeline enters in an operative mode where for each new signal acquisition the steps mentioned above take place.
- A tailored model-free (i.e., threshold-free) change-detection mechanism to inspect changes in the acquired data stream. By “threshold-free” we mean algorithms that are able to estimate in an automatic way relevant thresholds as part of the online (predictive) model estimation phase. Thus, the thresholds are still part of the processing, but they are not more hard-coded as part of the algorithm definition as it happens normally. A novel stage named “fine tuning” in the processing pipeline depicted in Figure 5 has been introduced in order for the estimate prediction model to not overfitting. Further details will be provided in section 4.4.

The residual can be modelled as an independent and identically distributed random variable and it is thus suitable for further processing by some sort of Change Detection Test (CDT) Analysis algorithm. Once the change has been detected by the model-free (i.e., non-parametric) CDT, the Change-Point Method (CPM) comes into play. The goal of this step is to further reduce the occurrence of false positive detections. CPMs are statistical hypothesis tests operating on a fixed data with the goal to verify whether the data sequence contains *a change-point*, i.e., a time instant after which the data-generating process changes its probability density function. When the presence of the change is confirmed this method also provides an estimate of the time instant the change occurred, which is a very important metadata needed to aggregate different changes from the same device or inter-device in order to cluster them together (D2.3-Table8-R.UC3.17). In essence, the role of CDT is purely practical to ensure real-time performances on target MCU. Its role is to quickly identify the presence of a potential change/anomaly in the behavior of the incoming signal/datastream. Only when a potential has been detected by CDT, the CPM method is triggered to 1st confirm the change, and then, find the exact anomalous point in the sequence (like a precise zooming process).
A similar approach for the change detection can be seen in [11] introducing the Support Vector Method for detecting changes in a given set of data. This kind of method seems very promising since it is in line with Vapnik’s principle that states *“never solving a problem which is more general than the one we actually need to solve as an intermediate step”*.
The developed SW middleware library has been integrated into STM32 MCU device and its Device Drivers SW Stack in order to map the functionalities on a self-contained binary firmware coding the IHES Sensing Unit functionalities. To exploit the results of the computed analytics and facilitate integration in SEMIoTICS a MQTT client has been integrated as well. Both the data acquired from the sensor and the all the events triggered by the embedded analytics have been exposed with a designed protocol anticipated in Task 4.4, that will be detailed in D5.6 – “Demonstration and validation of IHES Generic IoT (Cycle 1)” as part of the IHES UC3 demonstrator integration activities. At a higher level, since it is not the scope of this deliverable to cover these aspects, the communication of the IHES Device is roughly defined by six different types of messages:

- HEART_BEAT: this message is sent intermittently with the purpose to indicate that an IHES Sensing Unit is (still) correctly connected to the system.

- RAW_DATA: this message contains information about the raw data detected in an instant of time by a sensor of the node. Also it contains as part of the payload the difference between the values predicted and those detected.
- CHANGE_STATE: through this message the IHES Sensing Unit notify about the analytics pipeline state changes that have occurred on a given sensor.
- CHANGE_DETECTED: during the validation analysis of the sensor changes: if the outcome is positive, this message is sent for the purpose to notify the detection of a local event by a given sensor at a given time.
- SENSOR_TRAIN: this is a command sent to the IHES Sensing Unit. It instructs the node to retrain a given sensor on a new acquired training set.
- NODE_RESET: this is a command sent to the IHES Sensing Unit. It triggers a complete SW reset on the node.

2.3.4. IIOT/IOT FIELD DEVICES SUPERVISED LOCAL ANALYTICS

This subsection presents supervised clustering algorithms suitable for IIoT/IoT Field Devices. In particular, the focus is given on algorithms that can be applied to evaluate gait data acquired by inertial sensors like accelerometers and gyroscopes. This focus is motivated by the fact that gait analysis is one of the key functionalities provided by the SARA e-health use case. In general, gait analysis can be used for health monitoring or to verify the efficiency of rehabilitation and to evaluate surgeries' success. The acquisition of different kinds of kinematic gait data can be used for different purposes: measuring joint angles, determination of gait events (e.g., initial contact, end contact) and determination of spatiotemporal parameters (e.g., stride time, gait velocity), clustering of subjects into different groups (e.g., pathological, healthy, fatigued) based on their gait characteristics. Instance-based regression methods such as Generalized Regression Neural Network (GRNN) and k-nearest neighbors (k-NN) can be used to estimate joint angles during gait using inertial data [12]. Support Vector Machines can be used to classify fatigue and non-fatigue gait of healthy subjects using data collected from an Inertial Measurement Unit (IMU) situated at the sternum during fatigue and no-fatigue walking conditions [13]. Support Vector Machines can also be used to classify symmetric and asymmetric gait patterns using RGB-D data collected from a camera on board of a Robotic Walker [14]. In this paper, a Support Vector Machine (SVM) was trained according to the strategy "one-against-all," using a soft margin (cost) parameter set to 1.0. The chosen kernel is the cubic kernel. Multilayer perceptron neural networks can be used to cluster pathological and healthy gaits into groups using data recorded from with an accelerometer placed at the lower back [15]. Multilayer perceptron neural networks can be implemented using FPGA technology [16]. The current availability of low-cost FPGA technology like MicroZed[8] or Vidor[9] boards make this option suitable for a gait analysis system placed on board of the Robotic Rollator, one of the field devices part of the solution developed by SARA use case in SEMIoTICS (UC2).

[8] http://zedboard.org/product/microzed
[9] https://store.arduino.cc/mkr-vidor-4000

3. AI ALGORITHMS FOR EMBEDDED INTELLIGENCE

3.1. AI Algorithms in SEMIoTICS

AI algorithms and more specifically the “Deep Learning” approach is becoming increasingly popular throughout the world of technology. Artificial Neural Networks are at the core of Deep Learning methods. They are not new but they became more popular in the mid-2000s after Hinton and Salakhutdinov published in 2006 a paper [17] explaining how we could train a multi-layered feed-forward neural network one layer at a time. From their introduction (in the modern form) in the mid ’80s, they needed around 30 years to become mainstream, as computers were not powerful enough and companies did not have large amounts of data to train them. Examples of the ANN approach related to the ML can be found in Figure 7 below.

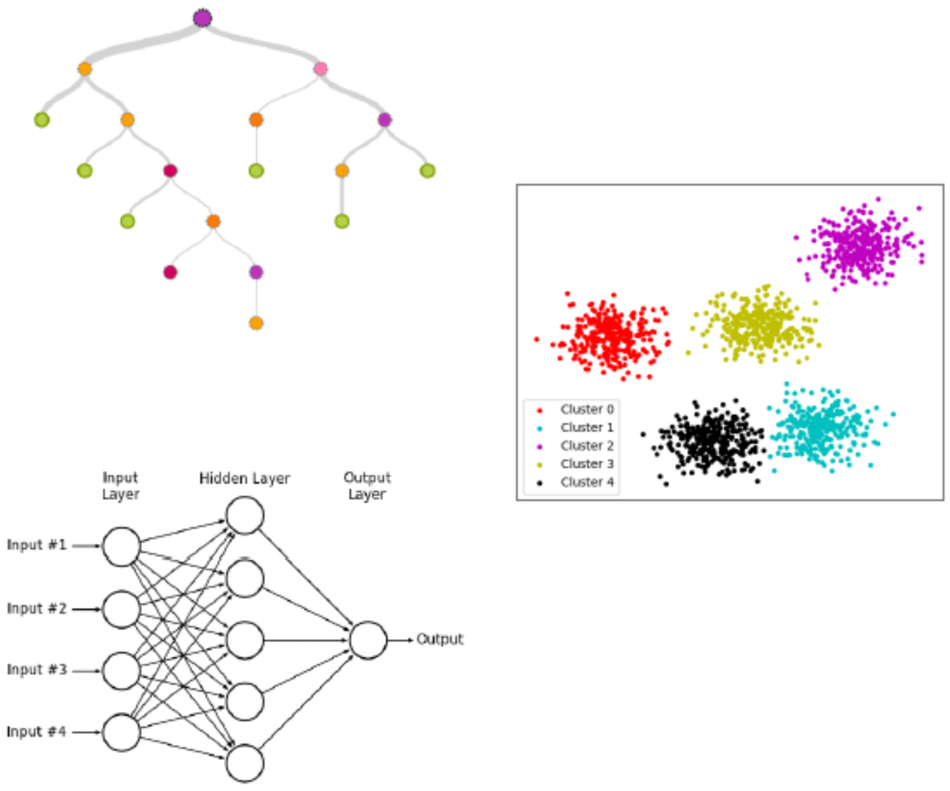


FIGURE 7: ANN APPROACH RELATED TO THE ML

More precisely, looking at Figure 8Figure 8 AI could be defined as a superset of all the studies where machines mimic the cognitive capabilities of humans. Typical examples are: interaction with the environment, knowledge representation and perception, learning, computer vision, speech recognition, problem solving, etc. AI is a heterogeneous topic that involves difference knowledge domains such as computer science, statistics, mathematics, etc.

ML is a sub-branch of AI: it is the field of computer science that gives computers the ability to learn without being explicitly programmed. It consists of algorithms that can learn and make predictions on data: such algorithms can be trained on past examples to build and estimate models. ML is usually employed where traditional programming is unfeasible. If trained properly, it should work on new cases.

Some typical approaches to ML are:

- Decision Tree Learning
- Clustering
- Classification
- Rule based learning
- Deep Learning

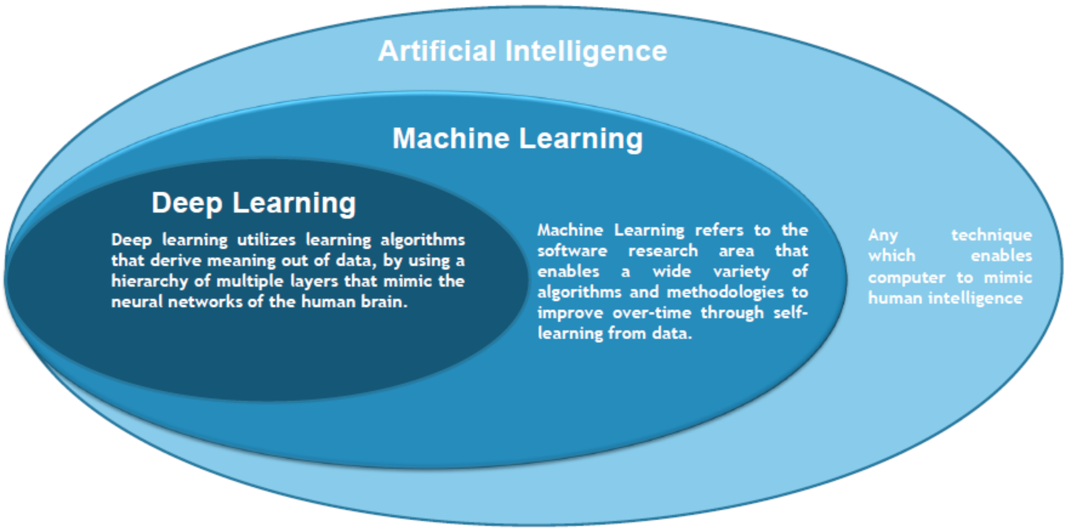

Artificial Intelligence
Machine Learning
Deep Learning
Deep learning utilizes learning algorithms that derive meaning out of data, by using a hierarchy of multiple layers that mimic the neural networks of the human brain.
Machine Learning refers to the software research area that enables a wide variety of algorithms and methodologies to improve over-time through self-learning from data.
Any technique which enables computer to mimic human intelligence

FIGURE 8: ARTIFICIAL INTELLIGENCE DOMAINS CLASSIFICATION

Decision tree learning is an approach aiming at estimating a model that can learn to predict discrete or continuous outputs by answering a set of simple questions based on the values of the input features it receives. As its name suggests, it is a tree-like graph with nodes representing the places where we need to take a decision/answer to a question; edges represent the answers to the question (usually it is a Yes/No answer, thus the tree is represented as a generic binary tree). The leaves represent the actual output or class label. They are used in non-linear decision-making processes, allowing at local nodes to expose them on surface, as simpler linear decision processes. Learning a decision tree from a dataset means to grow it by deciding which features to choose and what conditions to use for splitting the samples, along with knowing when to stop.

Clustering algorithms are useful for the detection of similarities. ML clustering algorithms do not require labels to detect similarities, so they do not need annotated datasets during the learning phase. For this reason, clustering algorithms are referred also as unsupervised learning. In the real world, unlabeled data are the majority of data and they are freely available (think e.g. about a sensor monitoring an environment). One law of ML is: the more training data are used, the more accurate our algorithm we expect to be. Therefore, unsupervised learning has the potential to produce highly accurate models: given a set of data observations (i.e., data points), we can use a clustering algorithm to classify each data point into a specific group. In theory, data points that “are similar” will lay in the same group (a group is a set of observed data points with similar properties and/or features); by contrary data points in different groups should have highly dissimilar properties and/or features.

A classification algorithm instead, depends upon labelled datasets: that is, the knowledge dataset used to train the network is properly labelled to allow the network during training to learn the correlation between labels and data. This is known as supervised learning. Typical application fields are to detect faces, recognize facial expressions, identification of objects in images, recognition of gestures in videos, detection of voices, transcription of speech to text, etc. Any label / class that humans can generate, any outcome that is relevant to solve a specific problem which correlates to data, can be used to train a neural network.

In a classification problem, the idea is to predict the target class by analyzing a labelled training dataset (i.e., a set of known observations). The goal of a classification algorithm is to find proper boundaries for each target class, in order to determine the class of unseen new observations and provide a correct class prediction probability for each possible class.

Rule-based machine learning (RBML) methods encompass any machine learning approach that identifies, learns, or evolves a predefined set of *rules*, where each rule specifies a subset of the input space. RBML uses partitioning methods for identifying subgroups of samples contained within the given training dataset. RBML methods are specializations of generic learning classifier systems [18], association rule learning [19] and artificial immune systems [20].

Finally, Deep Learning (DL) can be considered as a technique inspired by the human brain. It is said to be "*Deep*" because of the large number of layers and parameters, thus a lot of annotated data are required. The technology behind *Deep Learning* is a Neural Network composed by multiple layers stacked together. One of the fundamental practical challenge is to understand the exact information extracted by each layer. Each stack of neurons extracts higher level information, so that at the end they can recognize very complex patterns. Some experts are sometimes skeptical of this model because, even though it is based on well-known mathematical equations, we know little on the reasons why the defined deep neural network ultimately works.
This is a historical change in computer science. Before ANN, humans thought they were the best at designing code and rules, but now they have to accept that machines can beat them even in devising an algorithm. Machines programmed to recognize patterns with Deep Learning beat the old "Hand-Crafted-Rule-Based" algorithms.

3.2. AI Algorithms and the IIoT/IoT Local Analytics Revolution

Until recently, ANN and ML techniques were used only on the Cloud (i.e., Google Search or Facebook/Pinterest run ANNs/ML algorithms to identify user interests and propose news or ads). However, more recently they have been ported on automotive head units, industry 4.0 plants, smart building management systems and even drones or robots. This trend is going to spread further within the IIoT/IoT ecosystem and so every object will become not only *"Smart"* but *"AI-Smart"*. In particular, convolutional neural networks (CNN) will be of a paramount importance within the next years for the local analytics in IoT and IIoT domains, as the number of physically connected devices are going to exponentially grow to billions of connected devices in next 30 years as discussed in Section 2.2.

Once a neural network has been designed for a particular application, that network is ready to be trained, that is, the process of learning the network parameters weights. There are two training approaches – supervised and unsupervised learning. Supervised learning requires a mechanism to give to the network the desired output either by manually "grading" the network's performance or by providing the desired outputs expected from the inputs. Usually, large annotated datasets are required for proper supervised learning. On the contrary, unsupervised learning requires the network to make sense of the inputs without human help nor any a-priori knowledge on particular instances of the provided inputs. In SEMIoTICS, we plan to use both approaches to address different use case scenarios. The first approach usually is more suitable for classification problems whereas the latter usually can be adopted in self-learning appliances from time variant data streams to anomaly detection.

There are various ways to define a neural network architecture and to train it with specific (annotated) data in order to perform a specific task. Several DL tools have been developed in the past decade and they are widely

available as open source tools: Theano[10] (University of Montreal) and Caffe[11] (University of Berkeley) are the oldest libraries and frameworks available to design and train neural networks. In the last couple of years, they have been flanked and surpassed in terms of features and usability by others libraries/frameworks such as Keras[12], Tensorflow[13] (Google), CNTK[14] (Microsoft), MxNet[15] (Amazon), Matlab AI Toolbox[16], PyTorch, etc.
Neural Networks outperform solutions for people/environment behavior understanding and for modelling the outside world. The availability of large annotated datasets from one side, and the IoT revolution on the other, has made available new business opportunities and markets where AI enabled IoT devices are appearing and becoming "Smart".

Several Companies, ranging from Google to Facebook or from Amazon to Mobileye and Nvidia, are investing to provide affordable neural network solutions for various applications. With the advent of IoT, it is not convenient to delegate all system intelligence to the Cloud or to centralize the system brain in a computationally intensive, hugely power dissipating central units (CPUs), requiring more and more bandwidth to exchange data with the edge devices. *"AI-Smart enabled"* IoT devices allow filtering and processing data close to the source (i.e., the sensors) and off-load an important part of the processing from central units/cloud, also decreasing the required data transmission bandwidth and improving system scalability and responsiveness. Intel already confirms this trend, a primary company for Cloud based services, which is now relying on the Movidius IC solution[17] for Image/Video analysis based on CNN solutions, in order to offload the Cloud servers of more and more storage/computational power demanding processing required by the new applications.

Clearly, at this early stage of research applied specifically to IoT devices, not all CNNs could be ported to such memory/power/computation limited devices. As a simple example, CNNs for image processing are actually beyond the computational capabilities of any IIoT/IoT device based on simple MCU cores running in the range of tens of MHz (80 to 400 MHz Typically) with ROM/RAM sizes typically in the range 512 Kbytes/128 Kbytes. But there are some other application domains where ML algorithms could still play an important role as depicted in Figure 9 below.

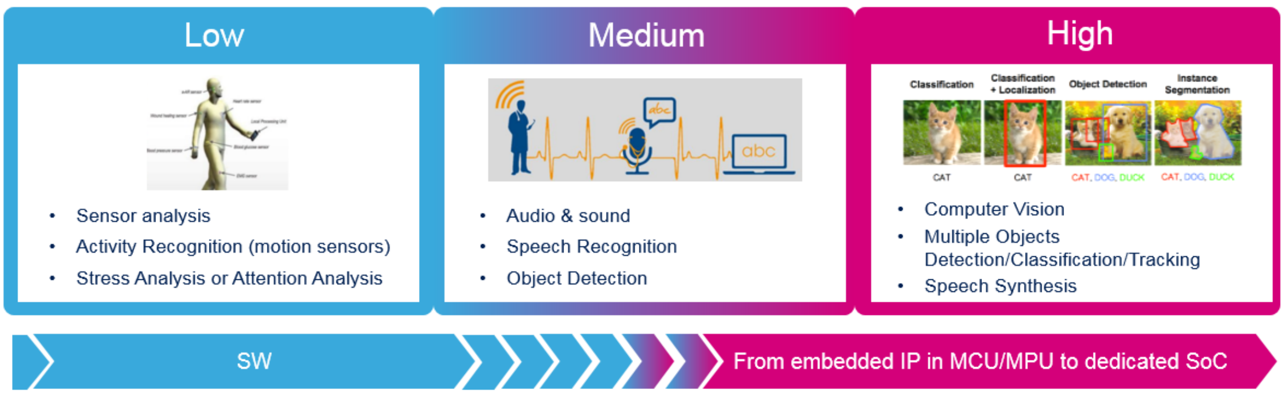


FIGURE 9: AI SOLUTIONS VS APPLICATION REQUIREMENTS[18]

[10] Theano DL Tool - http://www.deeplearning.net/software/theano/
[11] Caffe DL Tool - https://caffe.berkeleyvision.org/
[12] Keras DL Tool - https://keras.io/
[13] Tensorflow DL Tool - https://www.tensorflow.org/tutorials
[14] CNTK DL Tool - https://github.com/Microsoft/CNTK
[15] MxNet DL Tool - https://mxnet.apache.org/
[16] Matlab AI Toolbox - https://www.mathworks.com/campaigns/offers/ai-with-matlab.html
[17] Intel Movidius IC - https://www.movidius.com/
[18] STM32 solutions for Artificial Neural Networks - https://www.st.com/content/st_com/en/stm32-ann.html

In SEMIoTICS, we are interested in investigating the feasibility of deploying the set of appliances depicted in Figure 9, with a specific attention to the low-end size, as part of the Embedded Local Analytics Component. There are several reasons that motivates this focus. The main one is that in SEMIoTICS we are interested on the deployment of edge analytics up-to low power microcontroller IIoT/IoT nodes: this kind of devices are usually very limited in the available memory (few Mbytes maximum) and computational power (tens on MHz typically) and they are not (yet) equipped with any dedicated HW to accelerate AI/ML algorithms. Moreover, they are usually not equipped with any kind of vision camera since usually they are used to control physical actuators or process data from simple sensors. Thereby, they usually have plenty of environmental (temperature, humidity, etc.) and inertial sensor (mainly accelerometers); some devices include also microphones for simple audio applications and processing tasks. Addressing these potential applications allows to anticipate the needs and problematics of mapping these algorithms in real life scenarios, while still using a pure software-based solution even on these simple nodes, and anticipate on a real embedded system the edge computing paradigm. In fact, sensor analysis for simple tasks (e.g., predictive modelling, activity recognition, etc.) or some audio application (e.g., keyword spotting) could be addressed using “tiny”, specifically derived, AI/ML algorithms, where a simpler topology is defined from scratch, or by pruning complex NN models: these tiny models are composed by less layers, supporting a limited set of parameters. This simplified network topology requires smaller annotated databases in order to be trained, lighter training procedures and less memory usage by the final deployed models. A final aspect that allows this node level AI mapping is the general observation that AI/ML algorithms tends to have a peculiar “asymmetry”: they usually require powerful facilities (PCs, Server farms) to collect data and to train the models, but once trained, the trained model (the inference algorithm) is usually lightweight and simpler to run. The fact is that the first step is done during algorithmic development, offline, whereas actually only the inference algorithm derived from the trained model runs in the final target device. Also, it is interesting to note that depending on the specific scenario, AI/ML algorithms could address a task with higher accuracy and less computational and memory complexity figures compared to similar traditional designed algorithms as it is debated in [21]. Finally, another motivation for AI/ML algorithms in SEMIoTICS is that they effectively compress the information that has to be transmitted to the backend cloud. This decreases the load to the network and the cloud servers, and may end up conserving battery power. Although this is counter-intuitive, microcontrollers may need less battery power for AI/ML algorithms that considerably compresses information, than the transceiver chip would need to transmit the uncompressed, raw data to the backend.

3.3. Deep Learning Tools

Deep Learning Tools[19] are tools designed to support all phases of AI/ML algorithms, from the initial conception up to the final platform deployment. They are used by ML Engineers or Data Scientists to analyze data, train models, validate them, and, in some of them (e.g., Tensorflow), deploy them in production.
They aim at making the development of machine learning algorithms fast, easy, friendly (not restricted to only data scientists and mathematics) and to some extent, also interoperable.

- **Fast**: Good tools can automate each step in the applied machine learning process. This means that the time from the initial idea, the prototyping phase and the final platform deployment is greatly shortened. The alternative is to implement each capability yourself by hand-crafted code from the high-level model description. This can take significantly longer than choosing an off-the-shelf tool.

- **Easy**: You can spend your time choosing the good tools instead of researching and implementing techniques for your desired AI network model. The alternative is that you have to be an expert in every step of the (math) process in order to implement it. This requires research, experimentation in order to understand the techniques, and a higher level of engineering to ensure the method is implemented efficiently and with no bugs.

[19] https://en.wikipedia.org/wiki/Comparison_of_deep-learning_software

- **Friendly**: There is a lower barrier for beginners to get good results. You can use the extra time to get better results or to work on more projects. The alternative is that you will spend most of your time building your tools rather than on getting the desired results.

- **Interoperable**: not all tools are good for supporting all phases of the algorithm development. Some tools are very helpful in training the algorithm, others are more optimized for the deployment on specific platforms, and so on. Thus, an emerging need, not yet fully addressed but clearly understood, is that a good tool should allow seamless interoperability with other tools so that each phase can be carried out using the most convenient tool, according to the specific stakeholder's needs.

Interoperability is not yet a consolidated feature in today's tools: most of them are incompatible with each other. The community hasn't converged on standard formats and interfaces, and thus integrating tools across the entire workflow can be a very time-consuming task. Moreover, deployment is quite a tricky problem by itself regarding AI/ML tools: it means to be able ideally in no time and without any additional effort to put your newly designed and trained AI/ML algorithms in production by making them run on the final target platform device. This is usually referred as the *runtime environment* of an AI/ML tool. For example, IIoT/IoT services such as Amazon AWS and Microsoft Azure offer powerful proprietary runtime environments as remote services for ML algorithm deployment where the algorithms are often physically mapped on hardware accelerated GPUs and adapted to support this new computation intensive task. Ideally, a well-designed tool should support fast, efficient deployment on a heterogeneous set of platforms without involving the developer in dealing with specific platform boundaries or characteristics. This aspect has been mainly addressed today by virtualizing AI/ML algorithm deployment using powerful centralized datacenters facilities or mixed CPUs/GPUs architectures, and it comes at a cost: even higher complexity of the virtualization infrastructure, higher memory requirements and increased latencies in producing the expected results.

Recently, there is a trend in defining lightweight, limited runtime environments for embedded devices (e.g., the Tensorflow Lite runtime[20]), but they are still addressing the high side of those domain: usually the deployment is feasible as a suboptimal mapping relying on the availability of a platform housing powerful (embedded) multicore CPUs and GPUs like the ARM CortexA SoC. This runtime-virtualized environment-driven approach is clearly not suitable for the very constrained IIoT/IoT domain (i.e., mainly the MCUs domain). Moreover, there is not yet a consolidated standard approach for the deployment of those algorithms at this level of the infrastructure. In this respect, the SEMIoTICS project is aiming at defining a proper methodology and development flow for allowing fast deployment of lightweight ML algorithms at the IIoT/IoT Field Devices level equipped with low power MCUs. Further details are provided in section 3.3.6.
Anyhow, it is clear that the mainstream approach for developing AI/ML algorithms today is to rely on some DL tools for helping the whole development cycle of an algorithm or part of it. Over the past decade several AI/ML tools have been developed as open source software. Only few of them survive today and there seems to be a trend now where ML researchers, engineers, students and hobbyist tend to converge on three of them: Google Tensorflow, Keras and Facebook Pytorch as reported in Figure 10 below.

[20] https://www.tensorflow.org/lite

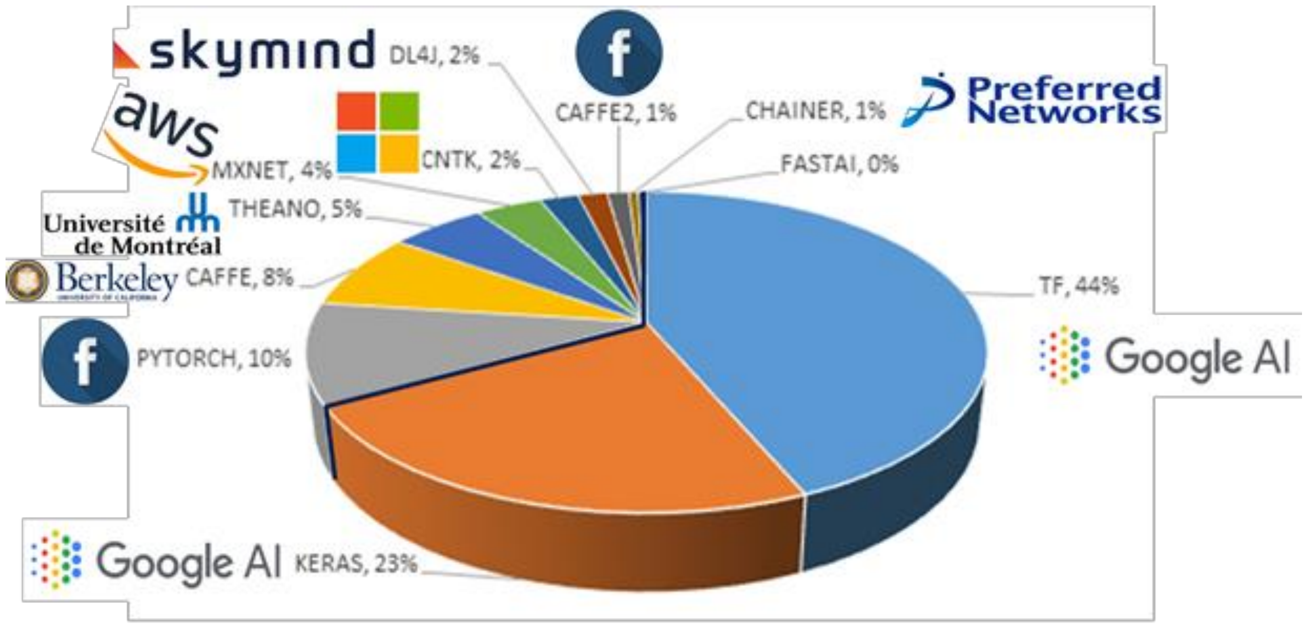


FIGURE 10: DEEP LEARNING TOOLS POPULARITY[21]

There are plenty of best-practice, tutorials and documentation on the web to start using any kind of DL tool. Abstracting from a specific tool, usually the steps required to develop (and deploy) a ML algorithm could be summarized in:

1. **Data collection** [22]: proper datasets usage is of paramount importance in the ML field. In the case of supervised learning, datasets should be annotated, while in the unsupervised case no dataset annotation is needed. Anyhow, even if not annotated they could be indeed costly to produce. A good dataset heavily impacts the accuracy of final algorithms since the network model is able to learn (and generalize) only from what it has observed. Feeding poor (or not relevant) data to a ML model during training often achieves a lower performance.
2. **Data pre-processing and conditioning**: pre-processing is another important step: it is not always the case that the desired ML algorithm receives as input the original data. In some case (e.g., in audio processing, inertial processing, etc.) some pre-processing algorithm should be designed in order to transform the input data in a more convenient data representation (this task is called feature extraction in the literature, as well). Typical examples are, for audio processing, the Fourier transform, and for inertial sensor, the gravity factor removal. These transformations are usually done to simplify the NN model by providing good conditioned data.
3. **Network model definition**: this step, together with the subsequent network training constitutes the heart of an ML algorithm. Defining a network model means to identify (based on some heuristics) the best set of basic layers and their concatenation (a network could be mathematically represented as a generic connected graph) in order to address the problem in a simple yet accurate way. This is usually referred as the identification of the right network topology and can be performed by experimentation through iterations of the training and optimization phases 4) and 5).
4. **Network model training**: once the topology is defined, the parameter set need to be sized as well: complex models and tasks usually map to larger parameter sets and increased model complexity, but higher accuracy, whereas small parameter sets mean in general lower accuracy but also "lighter" final inference loads. Since the "accuracy" of a network depends on the specific task (e.g., an autopilot algorithm for a real plane deserves a far higher accuracy than the one used in a flight simulator) there is always a trade-off between algorithm complexity and accuracy. Once the

[21] www.kdnuggets.com/2018/09/deep-learning-framework-power-scores-2018.html

parameters have been defined for a given model topology the actual training phase may happen. During this phase the algorithm starts to learn the task it is designed for: it needs labelled data if the algorithm is a supervised one, or annotated set of data samples for an unsupervised one. No matter if a dataset is annotated or not, it is a good practice to partition it in two parts: data used for the training ("training set") and data used for the algorithm evaluation "validation set" (see step 6). This is very intuitive to explain: an algorithm should never be tested using inputs that have been used for training it, otherwise it may be not able to "generalize" (i.e., to recognize unknown inputs when applied in a real-life environment).

5. **Network model optimization**: once the topology and the parameters have been tuned and identified as part of a successful training procedure, the model needs to be deployed on a real working environment. Since all devices have limited resources, including the most powerful ones, there are further steps to transform the original model representation into the actual deployed model. This is where the overall deployment phase plays a key role in order to ensure the algorithm works in real life situations, with optimal performances. There are several optimizations that could be actually implemented during deployment. They are specific to the model topology and platform design. Some common optimizations consist on parameters compression by any kind of scalar or vector quantization, internal network layers remapping as result of lowering procedures that are platform dependent, various parameters pruning techniques[22], etc.
6. **Network model platform deployment**: this step is the most critical one from the performance perspective. It is typically a task that is highly platform-dependent. The optimized model, with compressed parameters, is mapped to the target platform. Usually, this deployment on cloud-based services is highly automated and straightforward to be implemented: a *runtime environment* is usually available on the target that provides an abstraction from the actual available HW. The optimized mapped model, translated into a representation that the runtime can understand and run, is uploaded and instantiated in the real working environment. For example, the runtime hides implementation details regarding the actual mapping of the HW: the same NN runtime model may run efficiently on a GPU on a device A or on a NPU HW on a device B without any difference. However, in the case of limited resource devices (Gateway and IIoT/IoT MCUs nodes) it is not possible to have such runtime environment because this level of abstraction introduces several inefficiencies in the system: it could happen that the resources spent to run the model are more than the ones required by the ML algorithm itself. Moreover, on MCU devices a runtime is not feasible at all since it is not possible to handle or schedule any kind of dynamic memory allocation (required by a runtime): everything must be statically sized during compilation time for higher efficiency.
7. **Network model evaluation**: the final step of the process. The deployed model is fed, while running, with real data and its accuracy and performances are assessed to verify they conform to the requirements set for the use case in real life conditions.

The first steps mentioned above (roughly from 2 to 4) have been supported and they are available already on the majority of the most widely-adopted tools. However, the last part of the development flow is supported only on specific platforms/equipment supplied with a runtime facility. This is the case for almost all web services provided by Amazon or Microsoft at the Cloud level, but moving towards the leaves of the Cloud Infrastructure, i.e., to the local gateways or single node devices, no clever and integrated deployment flow is available today. In most cases, data scientists and ML engineers have still to hand-write and port their solutions to the physical devices. Thus, the deployment flow is always critical on IoT constrained devices, with no standard process or guidelines or tools to support it, even if it is a task that could be abstracted since it is not really related to the ML algorithms by themselves, but it is more platform bound. This abstract deployment flow has been conveniently represented in Figure 11, where it is clear that it could be somehow generalized and (partially) automated. The next subsections will shortly introduce each one of the main DL tools that are available online with a short summary of their more relevant features. In particular, in the last subsection the STM32 Cube.AI tool is presented: it is under investigation the possibility to use this tool in SEMIoTICS to support the deployment of ML algorithms on ST MCUs IoT sensing node devices, enabling highly scalable edge local embedded

[22] https://jacobgil.github.io/deeplearning/pruning-deep-learning

analytics at the field device level (D2.3-Table8-R.UC3.6). Parallel to the deployment of the local analytics on ST MCUs IoT sensing node, STM32 Cube.AI tool could be update to support the selected ML algorithm.

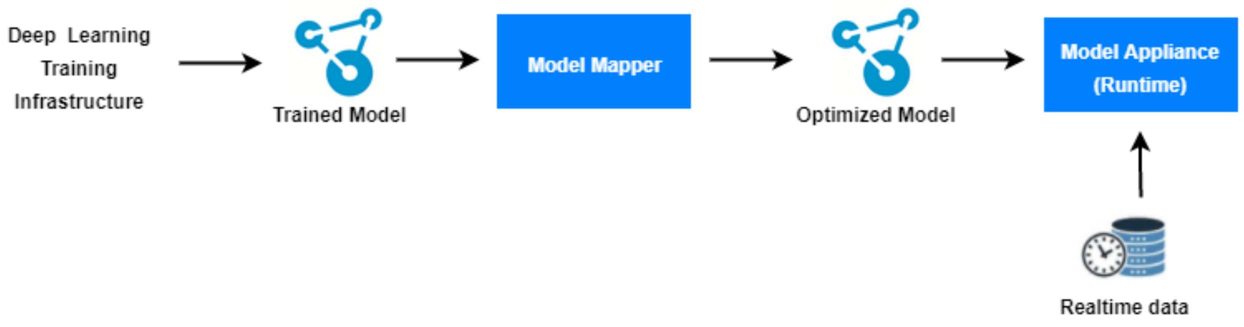

Deep Learning Training Infrastructure
Trained Model
Model Mapper
Optimized Model
Model Appliance (Runtime)
Realtime data

FIGURE 11: DEEP LEARNING TOOLS DEPLOYMENT FLOW

3.3.1. KERAS AND TENSORFLOW

Tensorflow is a free software library focused on machine learning created by Google. Initially released under the Apache 2.0 open-source license, Tensorflow was originally developed by engineers and researchers of the Google Brain Team, mainly for internal use. Tensorflow is currently used by Google for research and production purposes. Tensorflow is considered the first serious implementation of a framework focused on deep learning. It can run on multiple CPUs and GPUs and it is available on 64-bit Linux, macOS, Windows, and mobile computing platforms including Android and iOS.

At a high level, Tensorflow is a Python library that allows users to express arbitrary computations as a data flow graph. Nodes in this graph represent mathematical operations, whereas edges represent data that is communicated from one node to another. Data in Tensorflow are represented as tensors, which are multidimensional arrays. Although this framework for thinking about computation is valuable in many different fields, Tensorflow is primarily used for deep learning in practice and in research. It supports backend HW acceleration on dedicated GPUs or NPUs.

Keras is a high-level Python neural network API, supporting runtime backend environments such as Tensorflow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation of ML algorithms. Keras has been designed with these goals in mind: User friendliness (usable friendly APIs, Modularity (a NN model is expressed by a graph plugging fully configurable NN layers together), Extensibility (new layer modules could be defined in Python) and Python oriented (NN models are defined in Python and data processing could exploit other Python data science packages).

3.3.2. MICROSOFT CNTK

CNTK stands for *(Microsoft) Cognitive Toolkit*. It is an open-source toolkit, hosted in GitHub since 2012, for commercial-grade distributed deep learning. It represents NN models as directed computational graphs: the supported network topologies are deep convolutional neural networks (DCNN), Recurrent Neural Networks (RNN), and Long Short Term Memory (LSTM). CNTK has support for GPU acceleration. It could be used as a support library embedded in any Python, C# or C++ application or it could be used alone by defining models using its own model description language (BrainScript). It is the backbone of the Azure Machine Learning Cloud Services and the Cortana Windows OS assistant. CNTK supports backend HW acceleration on dedicated GPUs.

3.3.3. AMAZON MXNET

MXNET is hosted by Apache and it is used by Amazon. It is the sixth most popular deep learning library used in 2018. It is the core technology used in Amazon AWS Cloud Services. It has several features supporting commercial-grade distributed deep learning: device placement (it can specify where the actual NN model will run), multiple GPU training for increased scalability, etc. MXNET is a framework designed for accelerating generic numerical computations, but its specific focus is on accelerating NN models. For this purpose, it offers a wide set of optimized predefined NN layers. MXNET automates common workflows, so standard neural networks can be expressed concisely in just a few lines of code.

3.3.4. THEANO AND CAFFE

In 2007 the University of Montreal released Theano, which is the oldest widely adopted Python deep learning framework. During the last ten years it has lost much of its popularity and no major releases are planned, but the open source community still offers support and updates for it.
Caffe is probably the most widely used, C written deep learning tool and it is developed by Berkeley University (AI research group). It offers a convenient Python wrapper interface but the core of the tool is entirely written in C. This has become over time a bottleneck for tool usability. It is known as the first tool used by Google in 2012 to develop the *AlexNet 2012 DCNN Model*. Caffe is released under the BSD 2-Clause license, but today less and less developers are using it.

3.3.5. MATLAB AI TOOLBOX

The Deep Learning Toolbox provides a framework for designing and implementing deep neural networks with algorithms, pre-trained models, and applications. The user can run convolutional neural networks (ConvNets, CNNs, etc.) and long short-term memory (LSTM) networks to perform classification and regression on images, time-series, and text data. For small training sets, the toolbox can perform transfer learning with pre-trained deep network models (including SqueezeNet, Inception-v3, ResNet-101, GoogLeNet, and VGG-19) and models imported from Tensorflow, Keras and Caffe. To speed up training on large datasets, the toolbox can distribute computations and data across multicore processors and GPUs on the desktop in a parallel processing fashion, or scale up to clusters and clouds, including Amazon EC2 P2, P3, and G3 GPU instances.
Specifically, we developed a Matlab implementation for IIoT/IoT diagnosis within an IIoT/IoT botnet attack detection framework (see updated deliverable D4.9 for more technical insights about that). In terms of neural networks, we applied an auto encoder as developed in this Matlab auto encoder example[23].

3.3.6. STM32CUBE.AI TOOL FOR AI DEPLOYMENT ON IOT FIELD DEVICES

The STM32Cube.AI[24] plugin is an AI extension plugin for the STMicroelectronics STM32CubeMX Tool, supporting a wide set of STM32 Microcontrollers based on the 32-bit ARM® Cortex®-M architectures. STM32Cube.AI is an automatic tool to generate optimized C library code for STM32 MCUs from pre-trained neural networks. It guarantees interoperability with state-of-the-art Deep Learning Frameworks: this framework makes it is possible to deploy Artificial Neural Networks (ANN) and in particular CNN networks to any STM32 MCU powered board.

Nowadays, AI, ML and DL are mostly confined in the cloud, where unlimited computing resources seems to be available and evolving tirelessly. Systems with centralized intelligence are poorly scalable, slowly responsive, they require high power consumption, and they consume large communications bandwidth in the typical IIoT/IoT scenario exploiting hundreds of billions of sensors (Figure 3). On the contrary, moving to distributed intelligence systems with AI-enabled sensor nodes will provide higher scalability, lower latencies, optimized overall power consumption, more security and privacy.

Moreover, artificial neural networks can be used to classify signals or predict events from data provided by motion and vibration sensors, environmental sensors, microphones and other sensors, with higher accuracy

[23] Matlab auto encoder - https://www.mathworks.com/help/deeplearning/ref/trainautoencoder.html
[24] STM32Cube.AI - https://www.st.com/en/embedded-software/x-cube-ai.html

than conventional hand-crafted signal processing. But not all neural networks require powerful GPUs or complex SoCs with HW accelerators: neural networks can be implemented in SW on microcontrollers to be used efficiently in various applications. Automatic deploy of any pre-trained neural network developed with off-the-shelf Deep Learning frameworks (such as Keras, Caffe, Lasagne, etc.) to IIoT/IoT sensor nodes is now fast and easy using the new STM32Cube.AI.

In fact, the tool generates automatically optimized code to be used in STM32 microcontroller-powered intelligent devices at the edge, on the nodes, to be used in various IoT applications such as smart building, industrial, wellness, robotics and consumer products. The STM32Cube.AI plugin tool takes the pre-trained neural network model, generated by supported DL design frameworks, and then it automatically generates an optimized neural network C code library to run on the STM32 MCU, as bare metal FW. This flow provides the ability to test the pre-trained network model on the device in a very fast and efficient manner.

It offers a user-friendly semi-automatized support in developing AI solutions, assuming they are trained on resource unconstrained servers, but targeting resource-constrained STM32 MCUs. The STM32Cube.AI tool offers support for fast conversion of pre-trained neural networks into a memory efficient (ROM and RAM) and computationally (CPU cycles) fast optimized C NN library model.
In particular:

- The STM32Cube.AI architecture supports existing widely used DL tools and emerging technologies (such as Keras[13], TensorFlow[14], Caffe[12], Theano[11], etc.)
- The process from neural network model description to the related embedded SW (i.e., STM32 C Code FW) will be the most automatic possible

Stakeholders for this tool will be IoT *AI-enabled* developers, which can require a specific solution (for instance for Human Activity Recognition) to be retrained easily on annotated proprietary data and optionally having the possibility to optimize specific memory requirements without paying a noticeable accuracy penalty loss. These companies can be any vendor of AI-Smart devices. Then, in presence of an automatic ANN porting tool onto ST platforms, companies having a specific know-how on Neural Networks, developing their own solutions and requiring to import their ANN description in one of the most important DL frameworks, can benefit from the tool. The STM32Cube.AI tool can be used to easily deploy customer's solutions on specific ST platforms, such as X-

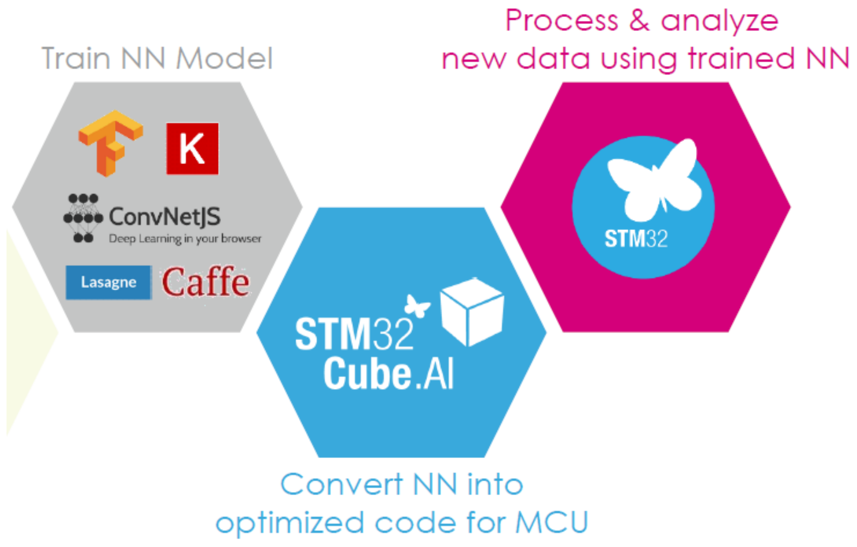


FIGURE 12: CUBE.AI TOOL PLUGIN

Nucleo Boards[25], BlueCoin[26] or SensorTile[27] development boards. The solution will need to address ST users that have already experience in deep learning and want to port their own solutions on ST platforms and customers that wants to exploit off-the-shelf solutions for their specific target applications. In particular, DL in SW on MCU can target many applications based on inertial sensors and audio processing such as Human Activity Recognition (HAR), Audio Scene Classification (ASC) or KeyWord Spotting (KWS), providing better support for fast deployment of these AI/ML learning models.

[25] https://www.st.com/en/evaluation-tools/nucleo-f401re.html
[26] https://www.st.com/en/evaluation-tools/steval-bcnkt01v1.html
[27] https://www.st.com/en/evaluation-tools/steval-stlkt01v1.html

4. LOCAL EMBEDDED ALGORITHMS IN SEMIOTICS

4.1. Overview

In the previous sections, a collective description of some algorithmic approaches, from SOTA analysis, has been presented: they were about regression models AI/ML methods and tools facilitating their design, anticipating the concept in SEMIoTICS about local embedded intelligence. In the following sub-sections, a further detailing of anticipated algorithms and some new one, defining according to the SEMIoTICS UCs scenarios and requirements, as declared in D2.2, D2.3, is provided. These algorithms will be the basis forming the final version of the Local Embedded Analytics ecosystem at field device level that will be used in the actual three demonstrators we are targeting. No matter the D4.10 SOTA analysis on algorithms and DL tools will allow to extend/combine new algorithmic ideas that may be needed to extend the UC definition. Thus, these methodologies declared in section 3, will be useful to provide assessed feedbacks during the implementation phase about the flexibility of the SEMIoTICS framework to accommodate new algorithms at the Field Device level architecture: one of the pillar objectives of the project.

An important subset of AI/ML algorithms, implemented as part of the Local Embedded Analytics ecosystem at the Field Device level, will be used to support the SEMIoTICS scenarios UC1, UC2 and UC3. A formal definition of the methodologies on which they rely on has been anticipated into sections 2.3.3 and 2.3.4, but will be fully developed this section 0.

4.1.1. RULE BASED LEARNING

As mentioned in Section 3.1 one of the approaches to ML is Rule-Based Machine Learning. Enabling reasoning, driven by production rules, appeared to be an efficient way to represent SEMIoTICS patterns. We use the Drools Rule Engine the theory of which is driven by the Knowledge Representation and Reasoning (KRR). KRR is the representation of knowledge in symbolic form whereas reasoning involves the actions that will be taken based on said knowledge. Figure 13 shows the basic Drools engine components where it is obvious that Rules and Facts are vital components. The Pattern Engine module enables the capability to insert, modify, execute and retract patterns at design or at runtime in the gateway involving components existing in the field layer. This is accomplished mainly by the Drools rules that are loaded in the Drools Engine, along with some auxiliary classes that were necessary to describe instances of the aforementioned components.

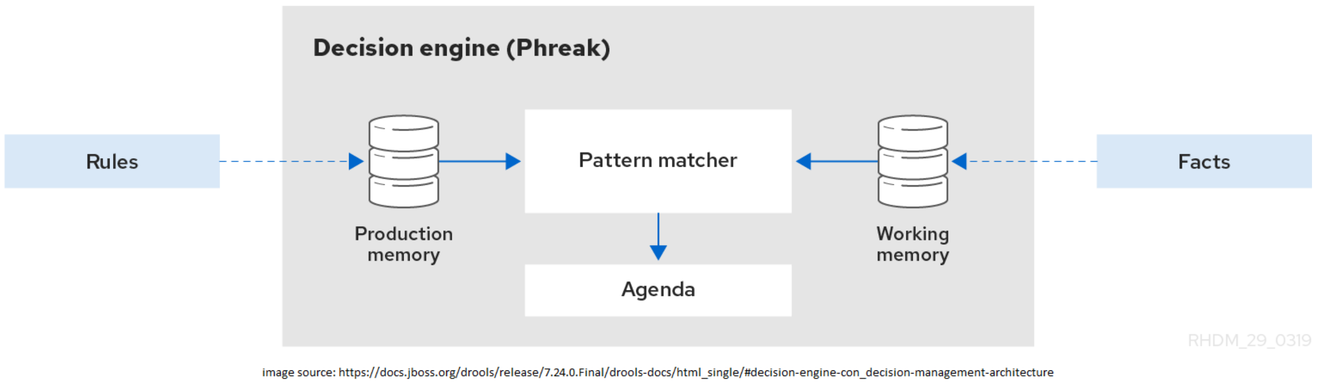

Decision engine (Phreak)
Rules
Production memory
Pattern matcher
Agenda
Working memory
Facts
RHDM_29_0319
image source: https://docs.jboss.org/drools/release/7.24.0.Final/drools-docs/html_single/#decision-engine-con_decision-management-architecture

FIGURE 13: BASIC DROOLS ENGINE COMPONENTS

The rule engine can leverage methods of forward chaining or backward chaining. Forward chaining is “data-driven” and thus reactionary. Facts are inserted into working memory, which results in one or more rules being concurrently true and scheduled for execution by the Agenda. On the other hand, backward chaining is “goal-driven”. The conclusion is part of the starting process which the engine tries to satisfy. If the initial conclusion cannot be satisfied, then the engine searches for conclusions that it can satisfy. These are called subgoals and are part of the current goal. It continues this process until either the initial conclusion is proven or there are no more subgoals. In Drools these subgoals are referred to as derivation queries. Figure 14 depicts the backward chaining process.

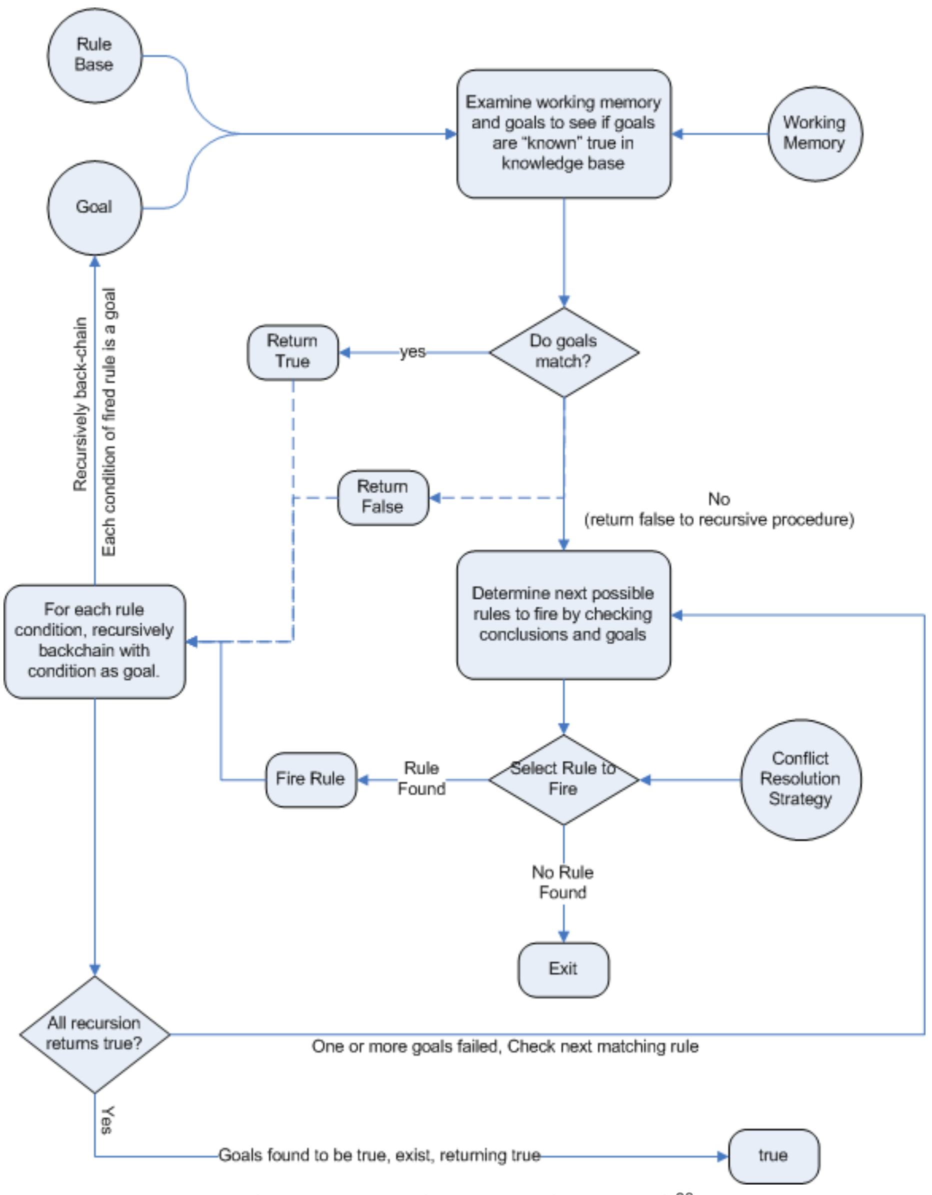

Rule Base
Goal
Examine working memory and goals to see if goals are "known" true in knowledge base
Working Memory
Recursively back-chain
Each condition of fired rule is a goal
Return True
yes
Do goals match?
Return False
No
(return false to recursive procedure)
For each rule condition, recursively backchain with condition as goal.
Determine next possible rules to fire by checking conclusions and goals
Fire Rule
Rule Found
Select Rule to Fire
Conflict Resolution Strategy
No Rule Found
Exit
All recursion returns true?
One or more goals failed, Check next matching rule
Yes
Goals found to be true, exist, returning true
true

FIGURE 14: BACKWARD CHAINING[28]

Although the Drools engine in Drools had used the Rete algorithm in the previous versions, the version used in PE module (6.5.0) works with an evolved algorithm. The said version uses the Phreak algorithm for evaluating the rules which has been proven to be more scalable and faster than its predecessors, which were Rete and enhanced Rete algorithm. Figure 15 shows how the Phreak algorithm tackles with the Rule Memory (Production memory). Essentially, it is composed of three layers, Node memory, Segment Memory, and Rule Memory.

[28] https://docs.jboss.org/drools/release/6.2.0.CR2/drools-docs/html/HybridReasoningChapter.html

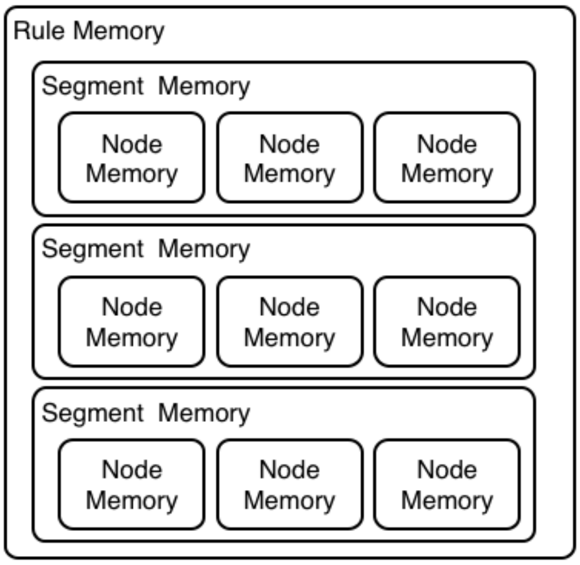


FIGURE 15: PHREAK THREE-LAYERED MEMORY SYSTEM

PHREAK propagation is set oriented. For any given rule that is under evaluation, it will visit the first node and process all queued insert, update and deletes. A set is formed with the results that is later propagated to the child node. The same process occurs in the child node resulting in the enrichment of the previous set. This goes on until the terminal node is reached creating a pipeline type effect for the rule under evaluation.
The rules for the PE module at the gateway, can be preinstalled or they can be pushed at runtime from the Pattern Orchestrator.

4.1.2. TIME SERIES CLUSTERING

In the case of Time Series Clustering, the execution of AI Algorithms derived from CNN and long short-term memory (LSTM) models aims to reduce the amount of data transferred toward the cloud: fixed-windows extracted from time series are processed on Field Devices and only encodings (i.e., algorithm outputs) are transferred to the cloud. Time Series Clustering is extremely important in the context of the SARA Healthcare solution (UC2): it is a practical example for showcasing the Local Embedded Intelligence component in practice, where the algorithm runs at the node level processing sensed data.

Within UC2 the problem of time series clustering arises in the context of Gait Analysis. Spatial-temporal characteristics of gait (e.g., gait velocity) can be used by experts for diagnosis and as indicators of falling risk.
The variables used for the analysis are the distances of the user’s legs from the Robotic rollator. These distances are measured by a laser range finder (LIDAR) mounted on the rollator (Figure 16). The two time series (i.e., left leg distance and right leg distance) are the inputs for the gait analysis algorithm.
The details of the ML algorithm developed for supporting the gait analysis task is provided in section 4.3.

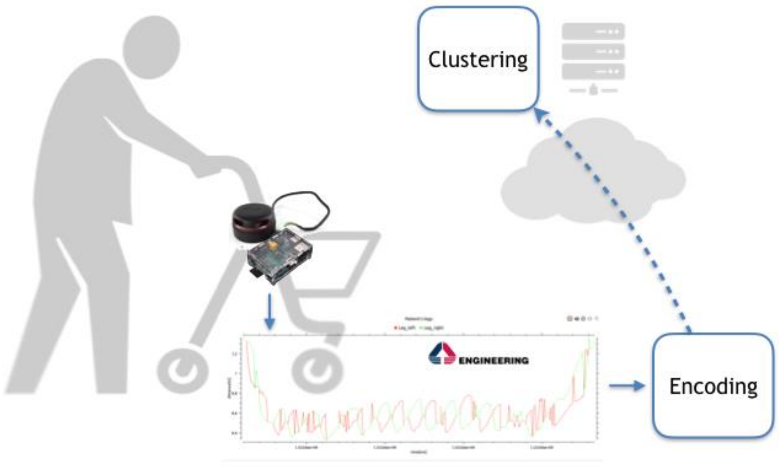


FIGURE 16: GAIT ANALYSIS SETTING IN SARA UC

4.1.3. TIME SERIES PREDICTION

In the UC3 IHES system, horizontal demonstrator, an unsupervised ML approach is used to estimate from a generic non-linear time series a model for predicting the signal from live acquired data directly. This prediction will be used to detect any relevant anomaly from the model learned. With this goal in mind, i.e. to manage generic non-linear complex dynamics on time series data, AI/ML models supporting the concept of recurrence have been investigated. Special attention has been given to Recurrent Neural Networks (RNN) models, as it seems they are a promising approach for deployment in time series modelling, and within these algorithms and models, the Echo State Network (ESN) models has been selected among the others, taking inspiration from the pioneering paper published in [23]. This type of AI/ML network model belongs to the reservoir computing family: it is a specific approach derived from RNNs and it is able to obtain results similar in term of accuracy to the ones achievable by traditional RNNs, but at a much lower complexity during actual training phase.

The ESN model that has been designed as part of the local analytics SEMIoTICS component is able to predict and model any kind of non-linear time series input, yet respecting the constraints of an embedded system equipped with very limited memory and an MCU unit running at ~80/100 MHz. The complete characterization of the algorithms developed for enabling this task is reported in detail in section 4.4.

By Local Embedded Intelligence algorithm, we mean any algorithm that takes roughly raw data and turns it into important information for the highest levels of architecture or for other software modules. They are considered, obviously, also all the algorithms supporting the neural / self-regressive models implemented in the MCU firmware.

According to the above definition, at least 4 algorithms have been developed intelligent within the firmware of the IHES Sensing Units. In particular:

- The algorithm that performs ESN online training on the predictor stage (see section 4.4)
- The Change Detection Test (CDT) algorithms in the detection stage (see section 4.5)
- The Change Point Method (CPM) algorithms in the validation stage (see section 4.6)

4.1.4. FEDERATED LEARNING

The use of centrally deployed ML is associated with challenges. Especially, for the training process, a single computing node must be able to access all training data, which requires massive storage and computing capacity. In addition, not all data can be used for training yet, due to aspects such as privacy, trade secrets and the necessary bandwidth for data transfer from remote devices are not taken into account.

Federated Learning (FL) is an ML approach that addresses these issues. It is based on the concept of bringing the code to the data, instead of the data to the code. A well-known application that already implements this approach is the *Google Keyboard GBoard* for smartphones, where the goal is to achieve the best possible "next

word prediction". Therefore, a model with a neural network is trained. This training takes place locally on the smartphone and with user-specific training data sets. The keyboard has already been installed more than a billion times and thus offers an enormous amount of training data, which leads to a very good result. In addition, the user's privacy is protected because the data never leaves the device. Only the model is exchanged.
The FL approach allows even more data to be used for machine learning purposes, which leads to better models. The influence of artificial intelligence or machine learning will thus increase significantly in the future in a wide variety of areas and will play a decisive role in shaping progress in research and industry. An algorithm based on FL approach for the specific UC1 scenario is presented and characterize in section 4.7.

4.2. Statistical Algorithms

4.2.1. ALGORITHM DESCRIPTION

In the current section, a detailed overview of the statistical-oriented approach for time series anomaly detection based on forecasting is provided. More specific, we adopt an autoregressive integrated moving average (ARIMA) model [23] (see Section 2.3.1). We assume that a time-series is obtained through a set of IoT sensors (in our scenario environmental data such as temperature, pressure, humidity etc. can be recorded) and the focus is given in predicting the next time-sample as well as determining if it constitutes an anomaly depending on the past data and a confidence interval rule.

Here, we assume a univariate discrete time-series (series in a chronological order) of the form

$$y_t : t = 1, 2, \dots, n, n+1, \dots$$

where n is a positive integer number. The time samples/measurements correspond e.g., to hourly, daily, weekly, monthly, quarterly, yearly etc. observations.
ARIMA(p, d, q) model can be used to approximate time series data. The initial letters ARIMA correspond to the so-called autoregressive integrated moving average model, with the first parameter p denoting the order of the auto-regression, which is the regression of the same variable, at lagged values. A p-th order AR(p) process y_t can be expressed as

$$y_t = c + w_1 y_{t-1} + w_2 y_{t-2} + \dots + w_p y_{t-p} + \varepsilon_t \ ,$$

where y_t is the stochastic process, c is the average of the changes between consecutive observations (if c is positive then the average change is an increase in the value of y_t, and thus y_t will tend to drift upwards, whereas if c is negative then y_t will tend to drift downwards), w is the weight of the correlation coefficients that is multiplied with the lagged values of y_t, and ε_t is an additive error term which is considered as white noise. The error term ε_t represents everything new in the series that is not considered (or cannot be modelled) by the past values.

The second parameter d of the ARIMA model denotes the amount of differencing that is needed, if necessary, to achieve stationarity (in other words, it corresponds to the integrated part of the model). A first order of differencing on the model can be written as

$$\delta y_t = y_t - y_{t-1}.$$

The third ARIMA model's parameter q corresponds to the moving average which is a linear combination of it's q past error terms written as

$$y_t = c + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \dots + \theta_q \varepsilon_{t-q}.$$

It is structured similar as the autoregressive part above, except the fact that regression is performed on the error terms, rather than the actual observations of y_t. Thus, if we combine the auto regression (AR), integration, and moving average (MA) parts (we assume that a possible differentiation is made first), then the AR and MA equations can be expressed as

$$y_t = c + w_1 y_{t-1} + w_2 y_{t-2} + \dots + w_p y_{t-p} + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \dots + \theta_q \varepsilon_{t-q}.$$

Akaike's information criterion (AIC) can be used in determining the order of an ARIMA model, i.e., selecting the best parameters (p, d, q). It can be written as

$$AIC = -2\log(L) + 2(p + q + k + 1),$$

where p is the auto regression parameter, q is the moving average parameter, L is the likelihood of the data (usually, an initial segment of the time series can be used to apply AIC-based model order selection), k is the number of model parameters, with $k = 0$ when $c = 0$ and $k = 1$ when $c \neq 0$.

When the time series behavior appears to be seasonally oriented, Seasonal ARIMA (SARIMA) model can be used to better model the time series. More specific, a SARIMA model is essentially an ARIMA model that includes seasonal terms, and thus it is a more effective model than classical decomposition allowing dynamic changes in the periodic component. The seasonal ARIMA$(p, d, q)(P, D, Q)_m$ is given by

$$\text{ARIMA} \quad \underbrace{(p, d, q)}_{\substack{\text{non-seasonal part}\\ \text{of the model}}} \quad \underbrace{(P, D, Q)_m}_{\substack{\text{seasonal part}\\ \text{of the model}}}$$

where m is the seasonal period. The seasonal part of the model consists of terms that are similar to the non-seasonal components of the model, but involve backshifts of the seasonal period. For example, an ARIMA$(1,1,1)(1,1,1)_4$ model (without a constant) is for quarterly data ($m = 4$) and can be written as

$$(1 - w_1 B)(1 - W_1 B^4)(1 - B)(1 - B^4)y_t = (1 + \theta_1 B)(1 + \Theta_1 B^4)\varepsilon_t,$$

where the additional seasonal terms are simply multiplied by the non-seasonal terms (for more details see in [23]).

4.2.2. VALIDATION AND TESTING

During the first simulations phase, we use open environmental data downloaded from the following repository:

- https://archive.ics.uci.edu/ml/datasets/Beijing+Multi-Site+Air-Quality+Data

This dataset includes hourly air pollutants data from twelve nationally-controlled air-quality monitoring sites. The air-quality data are from the Beijing Municipal Environmental Monitoring Center. The meteorological data in each air-quality site are matched with the nearest weather station from the China Meteorological Administration. The time period is from March 1st, 2013 to February 28th, 2017. Missing data are denoted as N/A. The attribute information is as follows

No: row number
year: year of data in this row
month: month of data in this row
day: day of data in this row
hour: hour of data in this row
PM2.5: PM2.5 concentration (ug/m^3)
PM10: PM10 concentration (ug/m^3)
SO2: SO2 concentration (ug/m^3)
NO2: NO2 concentration (ug/m^3)
CO: CO concentration (ug/m^3)
O3: O3 concentration (ug/m^3)
TEMP: temperature (degree Celsius)
PRES: pressure (hPa)
DEWP: dew point temperature (degree Celsius)
RAIN: precipitation (mm)
wd: wind direction
WSPM: wind speed (m/s)
station: name of the air-quality monitoring site

We focus on temperature data expressed in degree Celsius. Figure 17 depicts a time-series corresponding to temperature data.

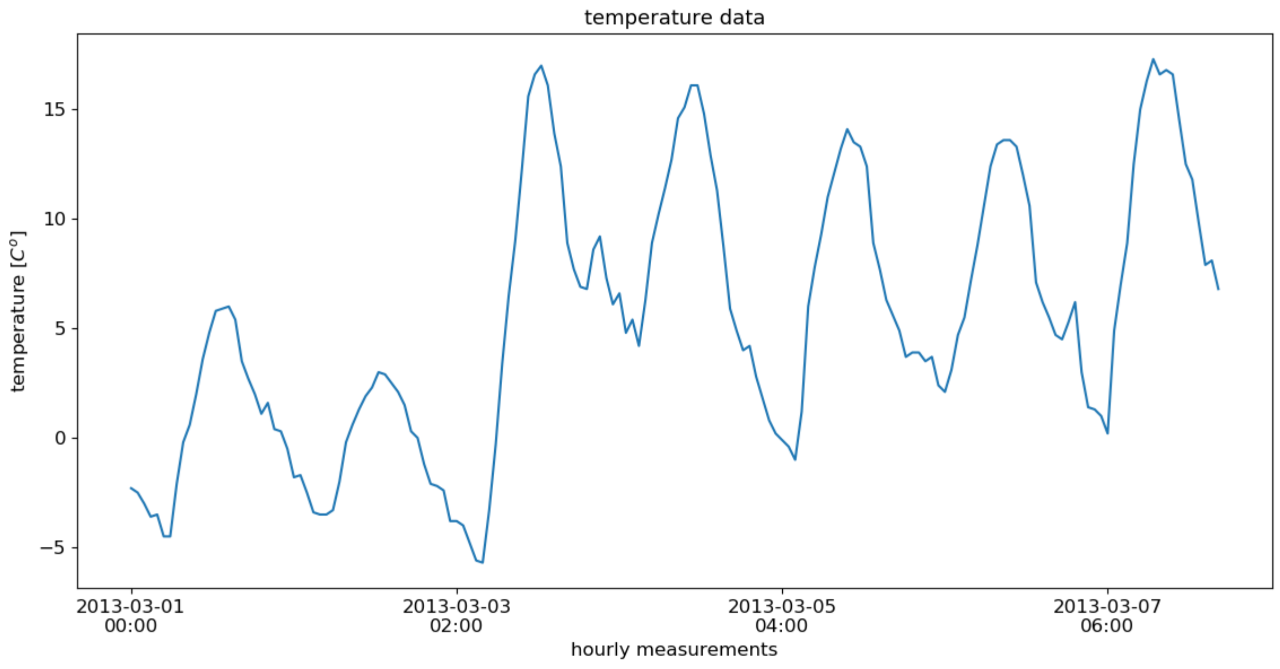


FIGURE 17: TIME SERIES (TEMPERATURE DATA)

We can observe that the autocorrelation function in Figure 18 has a seasonal structure, i.e., a periodic structure appears about every 24 samples, meaning that a seasonal ARIMA model can be adopted in our time series corresponding to a daily season (24 samples).

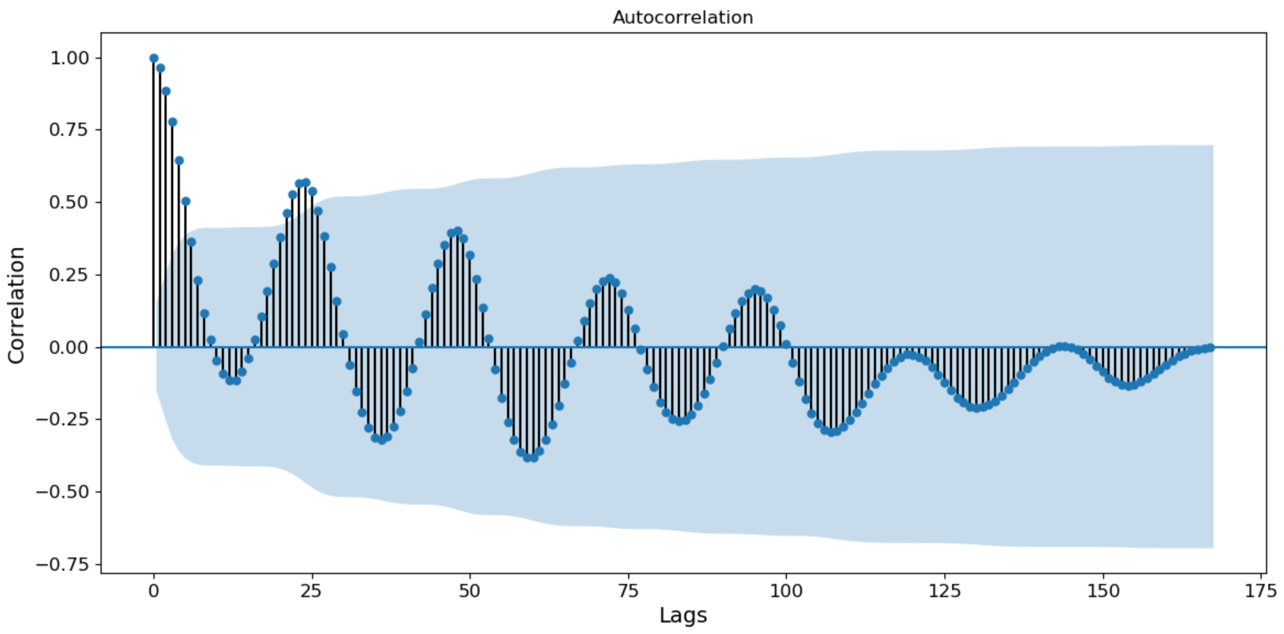


FIGURE 18: AUTOCORRELATION FUNCTION OF THE TIME SERIES DEPICTED IN FIGURE 17

Figure 19 shows a more detailed trend, seasonal and residual decomposition of the time series.

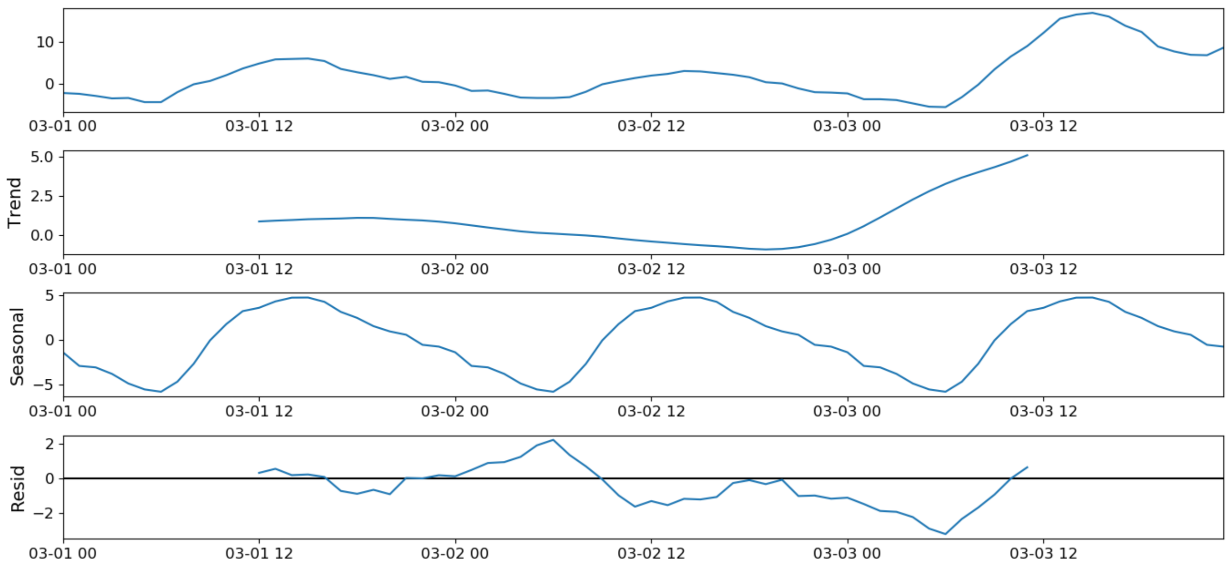


FIGURE 19: TREND, SEASONAL, RESIDUAL COMPONENTS OF THE TIME SERIES DECPICTED IN FIGURE 17

Next, towards evaluating the accuracy of the aforementioned model, we use the first 96 samples (i.e., 96 hourly observed temperature values) of the time series shown in Figure 17, in order to fit the seasonal ARIMA model with seasonal parameter $m = 24$. AIC-based model selection is performed using the forst 96 time series samples, and thus after an iterative process the best model corresponds to $(p, d, q,)(P, D, Q)_{24} = (1,1,1)(0,1,1)_{24}$ with an achieved $AIC = 64.84$.

The best selected model, is visually inspected in Figure 20. In specific, the basic goal is to ensure that the residuals of the best selected model are uncorrelated and normally distributed with zero-mean. In this case, our model diagnostics suggests that the model residuals are normally distributed based on the following remarks:

- The top right plot, depicts that the kernel density estimation (KDE) curve (orange solid line) follows closely the normal distribution (green solid line). This is a good indication that the residuals are normally distributed.
- The QQ-plot on the bottom left shows that the ordered distribution of the estimated residuals (blue dots) follows the linear trend of the samples taken from a standard normal distribution. So, this is a strong indication that the residuals are normally distributed.
- The residuals over time (top left plot) do not display any obvious seasonality and appear to be white noise. This is confirmed by the autocorrelation (i.e., the correlogram) plot on the bottom right, which shows that the time series residuals have low correlation with lagged versions of itself (all the correlogram values after the first lag are located within the confidence interval boundaries-shaded region).

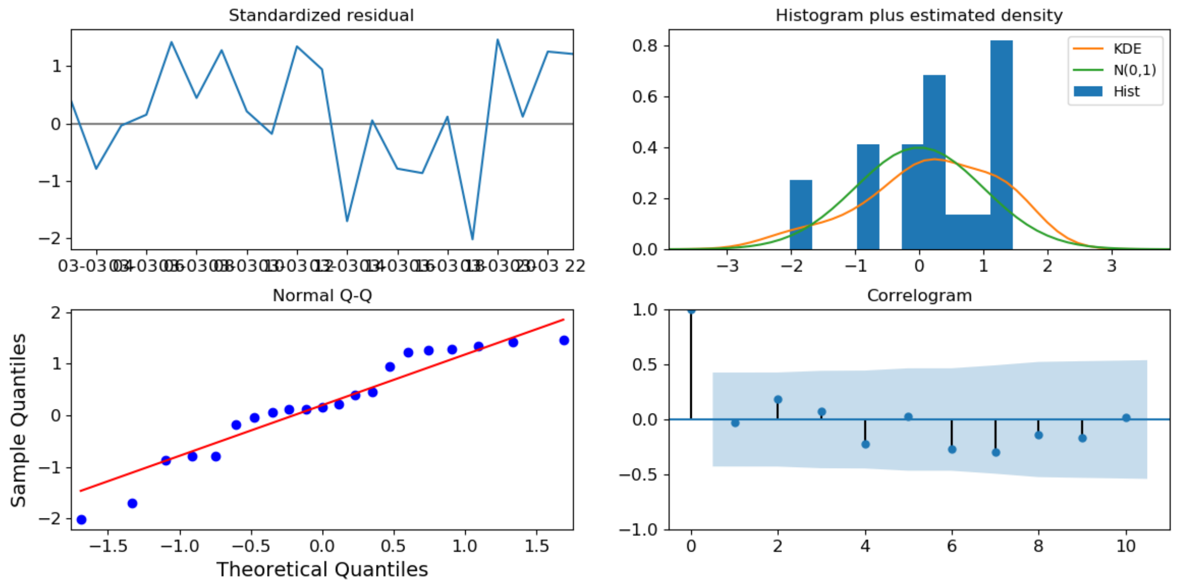


FIGURE 20: BEST MODEL SELECTION DIAGNOSTICS, BASED ON THE ESTIMATED RESIDUALS STATISTICS

In Figure 21, we can see the modelled time series, i.e., the reconstructed version based on the estimated seasonal ARIMA model $(p, d, q,)(P, D, Q)_{24} = (1,1,1)(0,1,1)_{24}$.

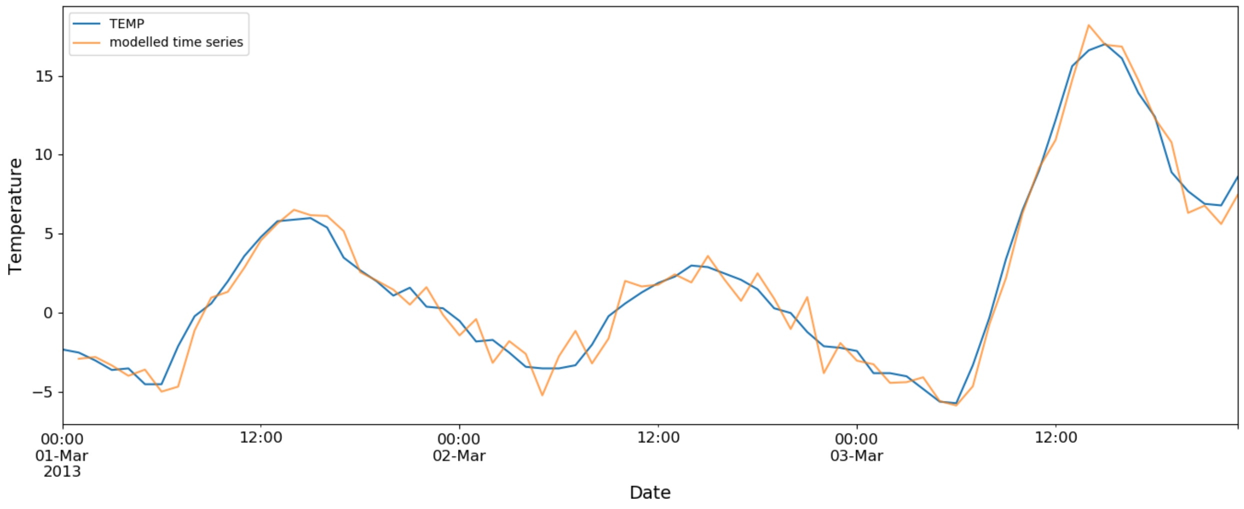


FIGURE 21: MODELLED TIME SERIES (ORANGE LINE) VS. THE ORIGINAL "TRAINING" SAMPLES (BLUE LINE)

Based on the estimated seasonal ARIMA model, we forecast the next 48 hourly temperature values. Figure 22 shows the predicted values against the actual temperature values.

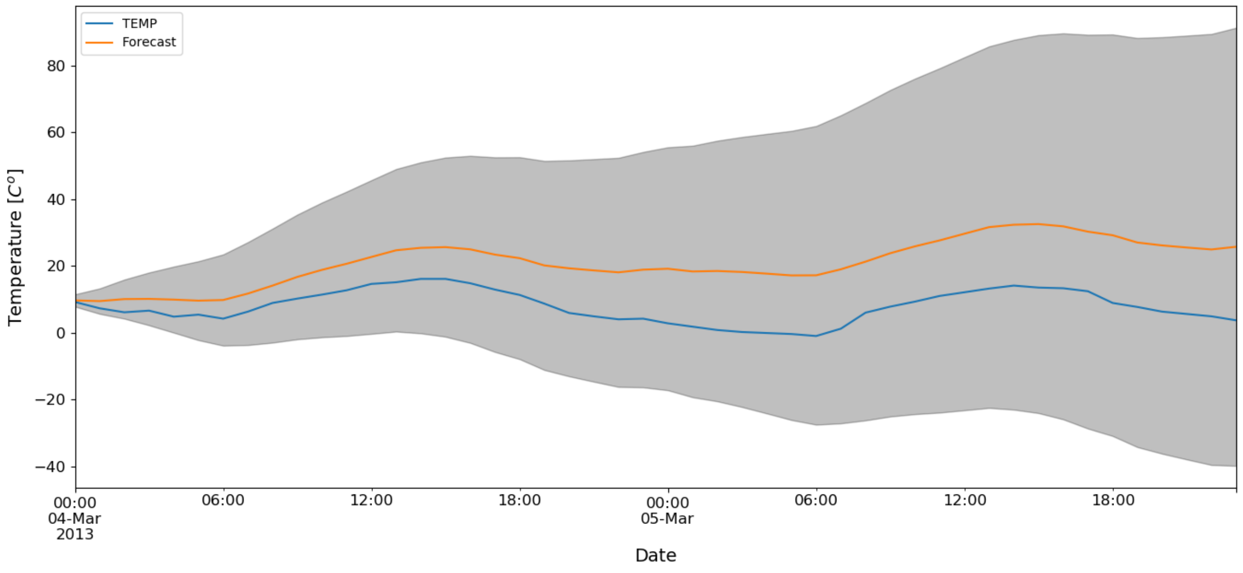


FIGURE 22: PREDICTED VALUES ('FORECAST') VS. ACTUAL TEMPERATURE VALUES ('TEMP')

The shaded region indicates the 95% confidence interval. It is obvious that the forecast accuracy is very good. This can be reflected into the mean squared error (MSE) which is MSE=237.82, and the root mean squared error (RMSE) which is RMSE=15.42.

Following the same algorithmic procedure, we use air pressure data samples (obtained in an hourly basis) in order to evaluate the performance of the method in a different type of data. Figure 23 below depicts the forecast against the actual air pressure data samples.

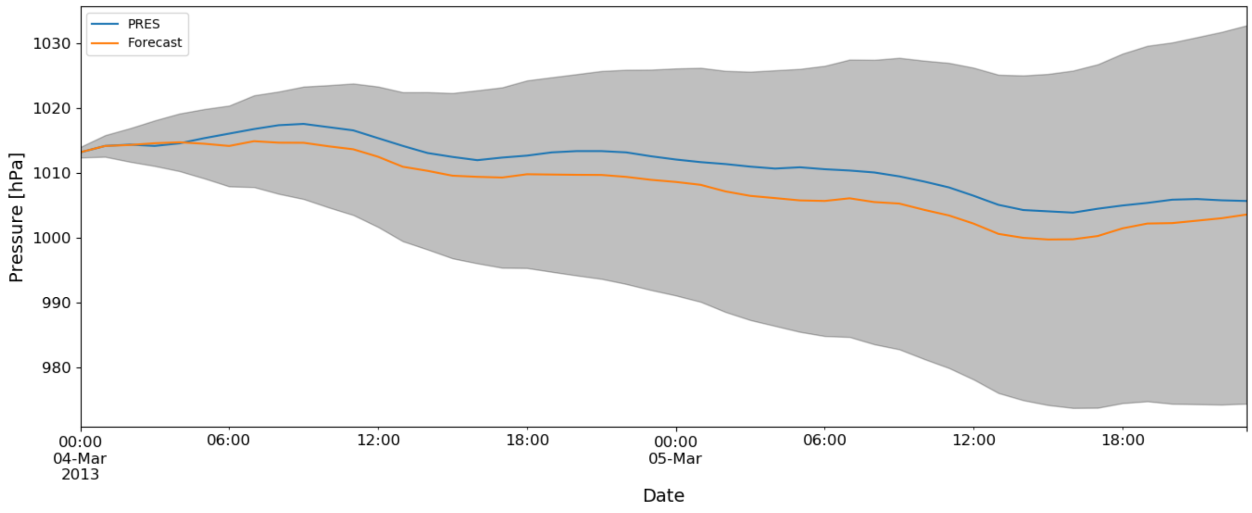


FIGURE 23: PREDICTED VALUES ('FORECAST') VS. ACTUAL PRESSURE VALUES ('PRES')

Again, the shaded region indicates the 95% confidence interval and it is obvious that the forecast we achieve a good prediction accuracy. This can be reflected into the mean squared error (MSE) which is MSE=49.8, and the root mean squared error (RMSE) which is RMSE=7.06.

4.2.3. DEPLOYMENT SCOPE

The forecasting algorithmic approach based on ARIMA modeling described in the previous section, can be applied within the SEMIoTICS framework for predicting anomalous data samples. The general anomaly detection rule is that when a predicted sample is located out of the confidence interval boundaries (see Figures 22-23) we can assume that the particular sample constitutes an anomaly. As a result, this statistical approach can be applied as a part of the SEMIoTICS framework when a predictive analytics module is needed towards a time series anomaly detector.

4.3. Time Series Clustering

4.3.1. ALGORITHM DESCRIPTION

As anticipated in section 4.1.2, the time series clustering algorithm described in this section is the key enabler for the gait analysis scenario.

In the developed algorithm, each time series is divided into fixed-sized windows using a sliding window. The objective is to find clusters in the fixed-sized windows, where each cluster represents a group of users (e.g., older adults, young people, etc.). UC2 will explore the use of the Deep Temporal Clustering (DTC) algorithm proposed in [24] for time series clustering/classification. The DTC algorithm makes use of CNN and LSTM networks. The DTC algorithm proceeds in three phases:

1. Reduction of data dimensionality and learning of dominant short time scale waveforms using CNNs
2. Further reduction of data dimensionality and learning of temporal connections between waveforms across all time scales using bidirectional LSTMs (Bi-LSTMs)
3. Clustering to split data into two or more classes

The first two steps are supposed to be executed locally on the Robotic Rollator whilst the last step is executed in the cloud.

The Figure 24 presents the main stages within the data flow of the Deep Temporal Clustering (DTC) algorithm. The stages within the rectangular box are those planned to be executed by the controller on board of the Robotic Rollator.

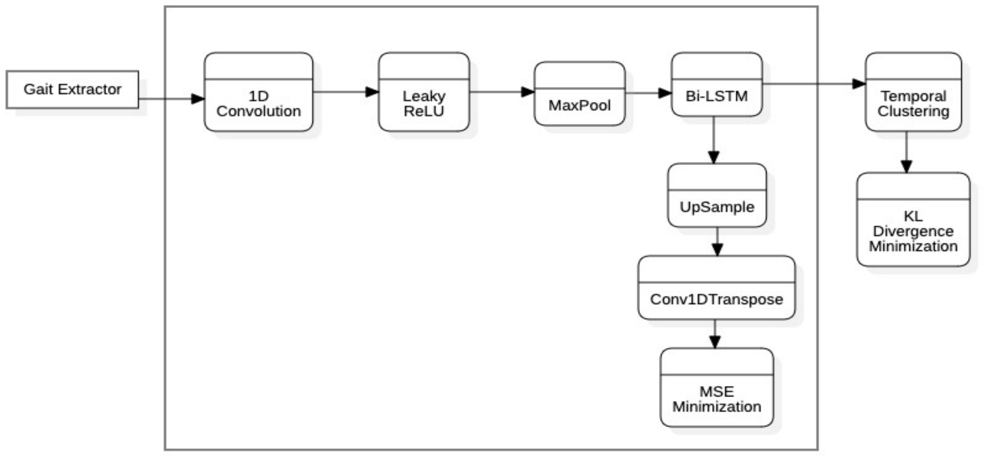


FIGURE 24: DEEP TEMPORAL CLUSTERING (DTC) DATAFLOW

The sequences processed by the DTC algorithm are generated by a Gait Extractor module which divides the time series generated by the LIDAR in subsequences each representing a gait event.
The first part of the algorithm proceeds along the following steps to obtain a latent representation of the time series generated by the Gait Extractor:

- Each sub-sequence from the Gait Extractor is encoded by a convolution operator having a leaky rectified linear unit (ReLU) as activation function.
- The encodings are further reduced by a MaxPool operator.
- The features extracted by the previous stages are temporally associated by means of a Bidirectional Long Short-Term Memory (Bi-LSTM) operator.

The latent representations extracted by the first part of the algorithm (i.e., the encoder) feed two subsequent pipelines:

- Reconstruction pipeline (i.e., UpSample, Conv1DTranspose) is aimed at reconstructing the input sequences from the latent representations. The reconstructed sequences are used by the Mean-Square Error (MSE) layer that tries to minimize the reconstruction loss by changing the parameters of the encoder (i.e., the parameters of the 1DConvolution and the Bi-LSTM layers).
- Clustering pipeline is aimed at clustering the latent representations of the input sequences. The clusters represent the input for a process that tries to minimize the clustering loss by minimizing the Kullback-Leibler (KL) divergence. Also this second minimization process acts thought the modification of the parameters of the encoder.

It is worth to note that the DTC algorithm optimizes both reconstruction and clustering loss using in an interleaved manner the two minimization processes mentioned above. This interleaved optimization of the two objectives has the goal to force the encoder to learn to extract spatial-temporal features that are best suited to separate the input sequences into clusters [24].

4.3.2. VALIDATION AND TESTING

As previously described, what we did first, was to take data from the LIDAR of the Robotic Rollator (RR). From these data, we distinguished two distinct points, which represent the distances of the left and the right legs respectively from the patient to the RR. The distinction between the left and the right leg is taken into account by the rotation of the LIDAR itself. After that, we have the two-time series given from the data representing respectively the left leg distance and the right one; those are the inputs for the gait analysis algorithm. In order to give a better view to the reader, in this paragraph, we consider a given dimension of numbers with one array that contains also eight arrays, each with one element, that we call "*data_in*". We developed the algorithm, using Keras. This one is an open-source neural-network library written in Python and capable of running on top of Tensorflow. What we are presenting now, is how we developed this algorithm; in particular, it will be shown in detail, with some part of code in Python language, the model chosen to develop it. Therefore, we created two **Sequential models**, the encoder, that it has its dimensions reduced in respect of the "*data_in*", and the other model, the decoder. This last one, given the output from the encoder's model, it will return an output that it has to be the same of the initial one: "*data_in*". The creation of the two models, it is possible by passing a list of layer instances to the constructors. Thereby, we need to define the two models, calling the "keras.models" library, following the part of code in Python, for the initialization:

```
from keras.models import Sequential
encoder = Sequential()
decoder = Sequential()
```

The first layer instance for the constructor of the encoder's model is **Conv1D**. This layer creates a convolution kernel that is convolved with the layer input over a single spatial (or temporal) dimension to produce a tensor of outputs. Its parameters are: filters, that for our example is 1, because it is the dimensionality of the output space. Then, kernel_size equal to 3, it is the length of the 1D convolution window and, the last parameter, it is input_shape, i.e., 8 and 1, for sequences of one vector of 8-dimensional vectors, given from the dimension of "*data_in*".
The second layer is Keras' **LeakyReLU**, is an activation layer (Leaky version of a Rectified Linear Unit). The LeakyReLU has a non-zero gradient over its entire domain, unlike the standard ReLU function. Its parameter alpha, it is the negative slope coefficient. The next one is a sample-based discretization process of the max pooling operation for temporal data, called **MaxPool1D**. This has a parameter, named pool_size, as it is possible to deduce from the word itself, that it is the size of the max pooling windows.
Next, **LSTM** (Long Short Term Memory), unlike standard feedforward neural networks, LSTM has feedback connections. In our case, we give the value 3, that it is the dimensionality of the output space. **Bidirectional,** it is because we are considering LSTM, in both directions. The code for the encoder model is:

```
from keras.layers import Conv1D, LeakyReLU, MaxPool1D, Bidirectional, LSTM

encoder.add(Conv1D(1, 3, input_shape=(8, 1)))
encoder.add(LeakyReLU(alpha=0.5))
encoder.add(MaxPool1D(pool_size=3))
encoder.add(Bidirectional(LSTM(3)))
encoded_data = encoder.predict(data_in)
```

We called "*encoded_data*" the first output from the "*data_in*" given. We used this output for two different ways: locally for the validation and, a copy of it to the cloud for clustering, as described in the dataflow of Figure 24,

```
from keras.layers import UpSampling1D,Reshape,Conv2D

decoder.add(UpSampling1D(size=2))
decoder.add(Reshape((1, 12, 1)))
decoder.add(Conv2D(1, (1,5)))
decoded_data = decoder.predict(encoded_data)
```

described in the previous introductory section. So that, the output will be first sent to the cloud and then the code, previously described that is running on the RR, continue passing a new list of layer instances to the constructor decoder, as follow:

As for the encoder's model, we have to pass a list of layer instances to the constructors that it is decoder. "Upsampling" layer for 1D inputs (**Upsampling1D**). It repeats each temporal step size times along the time axis. The second layer is **Reshapes**, to reshape an output to a certain shape; aimed at reconstructing the input sequences together with the third, and last layer, for decoder is **Conv2D**.
In the end, what we except is that, "*decoded_data*" is the same of "*data_in*". This it will happen the first time, for this reason the code it is all inside a loop, in which the data will enter and the algorithm will start to learn and to improve. Inside this loop there is a third Sequential model, that it is the auto encoder. This one, it will be called just one time inside this loop, because it is needed to optimize the encoding. The code is:

```
autoencoder = Sequential([encoder,decoder])
autoencoder.compile(loss='mean_squared_error', optimizer='sgd')
```

Fitting this model with the "*data_in*" and the "*decoded_data*", the system return the loss and the accuracy, so that the system can reduce the errors. This it is acceptable for our case, because we can lose some initial data, instead of fixing some parameters inside the code.

4.3.3. DEPLOYMENT SCOPE

The algorithm it is still under reinforcement. One important feature, it is to avoid the overfitting and we have to check what it is happening, if we introduce different values, with the **Dropout** function on both the encoder's and decoder's models.
On the other side, for the clustering in the Cloud, we want to split data into two or more classes, as it is shown on Figure 25, in which we show just an example of clustering in the Cloud, using random data divided in three classes, with k-shape method and Tensorflow modules.

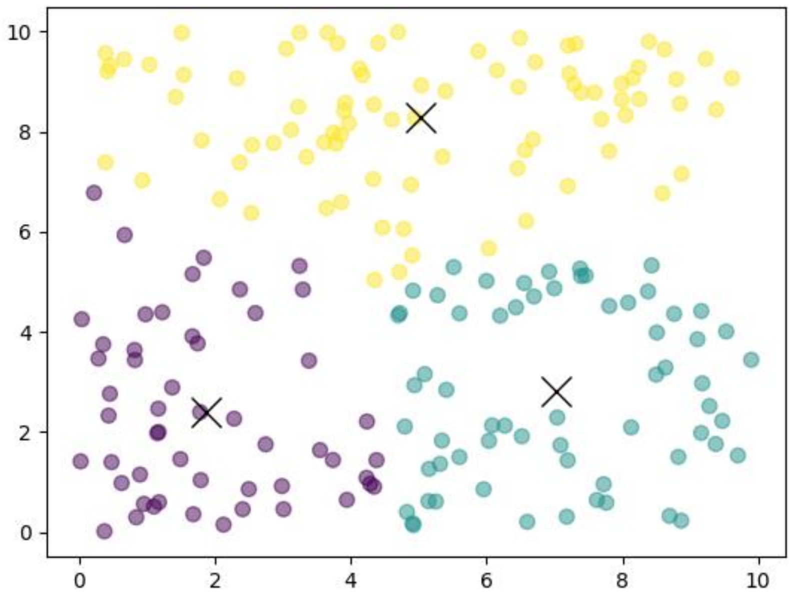


FIGURE 25: EXAMPLE OF CLUSTERING IN THE CLOUD, USING RANDOM DATA DIVIDED IN THREE CLASSES

This last part, representing the objective presented on the introductive part and presented on Deliverable 4.3, that it is to find clusters in the fixed-sized windows, where each cluster represents a group of users (e.g., older adults, young people, etc.).

4.4. Time Series Prediction

4.4.1. ALGORITHM DESCRIPTION

As anticipated in section 4.1.3, the problems that have been considered during the model definition are related to the presence of many heterogeneous sensors (D2.3-Table1-R.GP.1), tight constraints in term of memory and computation for each single node, and high scalability of the system. The Echo State Network (ESN) models, first introduced by Jaeger [25], are a type of AI/ML network architecture based on Reservoir Computing[29] (RC); RC is an innovative design and training paradigm for a special kind of recurrent neural networks. In particular, RC is defined by two main families: ESN and Liquid State Machines (LSM).
The term RC emphasizes that both techniques share the same basic fundamental idea, that is, the separation between the recurrent part of the network, the neuronal (or dynamic) reservoir (DR), for the non-recurrent part, and the output readout layer. This allows to split the overall training process of the entire RNN in two distinct phases: the training of the hidden reservoir layer (usually done offline), and the training of the output readout layer (online training). In particular, once the DR has been trained, the most common approach is to train the readout layer by linear regression or through an approach based on gradient descent. This avoids the use of back-propagation techniques on the entire RNN as it is in the case of back-propagation through time.

The input signals to the network guide the non-linear DR, producing an internal hidden dynamic state called echo response, which is used as a basis for reconstructing the desired outputs through a linear combination of the DR outputs. In other words, the DR layer could be considered as composed by features used as input for the output readout layer. This strategy has the advantage that the recurrent network within the DR is fixed, so it is not involved in the online training phase. The ESN are network models derived from the reservoir computing. ESN model has been selected as the algorithm used for signal model predication and residual computation in highly non-linear input data such as the one generated by an inertial sensor (i.e., accelerometer). The ESN model used for the implementation of the time series prediction task could be formally defined and visually represented as follows:

[29] https://en.wikipedia.org/wiki/Reservoir_computing

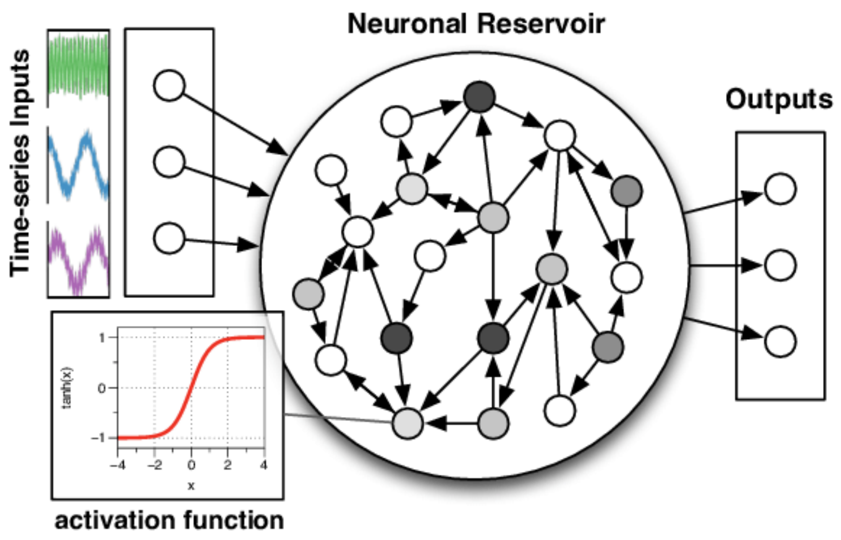


FIGURE 26: ESN MODEL FOR TIME SERIES[30]

The link between inputs and outputs allows the model to choose whether and to what extent to exploit the dynamics of the DR. Furthermore, the feedback link between outputs and DR has been made in order to avoid adding complexity and reducing the possible occurrence of instability problems as reported in [26].
The ESN embedded network model is specified by this model relation:

$$x(n+1) = f(\boldsymbol{W_{res}} * x(n) + \boldsymbol{W_{in}} * u(n) + \boldsymbol{b})$$

Where:

- $x(n+1)$ is the state at time $n+1$
- $\boldsymbol{W_{res}}$, $\boldsymbol{W_{in}}$ are respectively the DR dynamic neuronal reservoir and input weight matrices
- $\boldsymbol{b}$ represents the added bias factor
- $f()$ is the non-linear activation function of neurons
- The input and the hidden state at the time instant n are defined by $u(n)$ and $x(n)$ respectively

This non-linear model can be used to define a prediction at time $n+1$ given a set of n previous observations.
Part of the task 4.3 activities has been devoted to the proper identification and configuration of the ESN model parameters space. In this paragraph we detail the guidelines followed to identify this optimal configuration together with the theoretical concepts followed for the design of the ESN model. To overcome the HW constraints that MCU device has, several design choices have been made to simplify the theoretical model of ESN previously discussed, yet ensuring *its predictive capabilities*. The choice of parameters and constraints for ESN was made based on theoretical analysis supported from experimental evidences. Furthermore, the search for the best configuration is from the computational point of view expensive due to the number of parameters of the ESN embed and the amount of data needed to make this choice. So we define during the initial part of the activities an *ESN-configuration* module, written in python, that takes input from the live acquired data, and determine the space of the parameters (i.e., the SPACE from now on) and returns a configuration *bestESN*. The *bestESN* configuration is a parameters tuple (input layer, $\boldsymbol{W_{res}}$, $\boldsymbol{W_{in}}$, $\boldsymbol{b}$), of the ESN trained model that minimizes the root mean square error (*rmse*). The *rmse* is computed as:

[30] Courtesy of https://www.researchgate.net/figure/Echo-State-Network-ESN-In-the-typical-setup-the-inputs-are-fully-connected-to-a_fig1_263124732

$$rmse = \sqrt{\frac{1}{n} * \sum_{i=1}^{n} (R_i)^2}$$

Where n is the number of predicted samples and R_i is the residual (difference between real value and predicted value) computed at sample *i.*
The methodology implemented in the *ESN-configuration* module for the identification of the optimal *bestESN* configuration is presented in Figure 27 and has been derived by implementing the method suggested in [26], [27] and [28].

```
$min_{rmse} \leftarrow MAXNUM$;
$number_{trial} \leftarrow K$;
$i \leftarrow 0$;
forall the $c$ in $SPACE$ do
    while $i \leq number_{trial}$ do
        $training_{set} \leftarrow getTrainingSet(data)$;
        $testing_{set} \leftarrow getTestingSet(data)$;
        $esn \leftarrow configure(c.SR, c.IS, c.N_{res}, N_{in}, N_{out})$;
        $training(esn, training_{set}, c.B)$;
        $rmse \leftarrow testing(esn, testing_{set}, c.B)$;
        if $rmse \leq min_{rmse}$ then
            $min_{rmse} \leftarrow rmse$ ;
            $bestEsn \leftarrow esn$ ;
        $i++$;
return $bestEsn$;
```

FIGURE 27: ESN-CONFIGURATION MODULE ALGORITHM

We set initially the number of experiments *K* and the SPACE set of all possible solutions. For all the solutions and all the trials we iterate acquiring the training set, the testing set, configuring the spectral radius (*c.SR*), the input scale (*c.IS*), the number of neurons ($c.N_{res}$=50 in our case), the number of inputs (N_{in}=3 in our case) of the model, the number of outputs (N_{out}=1 in our case) of the model, and the bias factor (*c.B*). These hyper-parameters have been set during an initial characterization of the use case as an optimal trade-off between memory/computational complexity and the performances of the algorithm in the desired target working range as described later in this section.
The *configure()* step generate the ESN DR matrices according to the provided parameters and return the ESN model to train. The *training()* step is the routine used to train the configured ESN model. It takes as parameter the bias factor and the training set. The *testing()* routine computes the *rmse* value from the testing set. If the computed *rmse* value is lower than actual computed, the ESN configuration is kept in *bestESN*. At the end of the procedure the *bestESN* model configuration is selected as the model to be deployed at runtime for the prediction task on the MCU target device.
The dataset used to characterize the optimal parameter set of our ESN model is named the *NUC dataset*[31] composed of several live sequences acquired in different configurations from inertial sensors. It could be noted that the database does not require any annotation since we are designing a predictive network: thus, the expected predicted value at a given time *n* is by definition the value at time *(n+1)* which is known.

[31] The dataset consists of a set of data streams acquired directly from the accelerometer and gyroscope sensors, using the x-nucleo-iks01a2 expansion shield for the STM32 X-Nucleo Board. The data are acquired at a frequency 5Hz for a period of 70 seconds

4.4.2. VALIDATION AND TESTING

In this section a theoretical comparison between the ESN model previously described and more traditional Recurrent Neural Networks (RNN) models, with particular attention to the theoretical complexity of the models considering both in the training phase and the operational phase. A remarkable point, when it comes to non-linear learning mechanisms is the computational complexity of the training methods involved, especially if the execution of these models is placed on embedded systems. An analysis about memory and power consumption is provided as well as part of the experiments and characterizations done during the task activities.

In general, the RNN models are based on a gradient-descend-based learning algorithms for implementing the training phase. These algorithms require *O(n^3)* as spatial and temporal complexity [29], against one linear regression which is *O(n^2)* as time complexity and *O(n)* as spatial complexity (i.e., memory). The gradient descent approach is also well known for its slow convergence due to the low learning rate, a parameter used in the algorithm of training to determine how much the update step current affects weights, which typically takes a low value to avoid getting stack in local minima or instability problems.

Moreover, another problem that arises for this type of algorithms is the well-known widely studied gradient descent problem [30], to which there are various solutions in literature including LSTM architecture and general RC theory. Solving this problem would introduce additional complexity in the architecture, as in the case of LSTM networks as reported in the survey.

However, an ESN model does not suffer from this type of problematic behavior and the training methods used are particularly simple and inexpensive from a computational point of view.

Besides, it is stated that even if the overall performances are lower than the ones obtained by classic RNN, yet this difference does not justify the added complexity that RNNs have vs the ESN models [31].

Thus, we could argue that the theory of reservoir computing and in particular the ESN approach demonstrates from a theoretical point of view that it can be a valid trade-off to our goals of affordable real-time online training on heavily constrained MCU devices.

4.4.2.1. MEMORY COMPLEXITY ANALYSIS

For what concern the memory consumption, some formula has been derived from the analysis of architecture and training method, from which we derive the relationship between the number of neurons and the consumption of memory (in Kb) of the ESN model:

$$\begin{cases} SRAM = (N_{in} + N_{res}) * (DIM + N_{out}) \\ SRAM_{training} = N_{res} * (DIM + N_{out} + N_{res}) + N_{in} * (DIM + N_{out}) \\ FLASH = N_{res} * (N_{in} + N_{res}) \end{cases}$$

where:

- $SRAM$ is the static Random Access Memory (RAM) consumption during the runtime phase of the ESN model
- $SRAM_{training}$ is the static RAM consumption during the training phase of the ESN model
- $FLASH$ is the device Read Only Memory (ROM) flash memory consumption of the ESN model
- N_{in}, N_{res}, N_{out} is the number of input neurons, the number of reservoir neurons and the number of output neurons, respectively, while DIM is the number of samples of the training set

TABLE 2. MEMORY FIGURES

ESN Model – Memory figures (Accelerometer)	Kb
$SRAM$	15.65
$SRAM_{training}$	20.6
$FLASH$	5.32

The figures of Table 2 have been obtained by profiling the memory consumption of the firmware library containing the C mapping of the developed ESN model characterized as described in the previous sections.

4.4.2.2. POWER CONSUMPTION ANALYSIS

This section presents the results and figures about the energy and power measurements related to the complexity of the ESN model developed in this section. The tests were done on the *NUC dataset*, comparing the ESN model with a simpler auto regressive model (AR). The test setup used for the experiment provides that, after training the reference AR and ESN models (with a training set of 100 samples from the *NUC dataset*), the two models enter in predictive mode, where they continuously acquire real-time data readings from the sensor shield at the instant of time *n*, and the value *n+1* is instantly predicted by either the reference AR or ESN models. The average measurements of the instantaneous powers and energies are presented for both the training and prediction stages. In particular, for the ESN model, data acquired from the accelerometer where used, while the AR model has been tested with temperature, pressure and humidity data readings.
The methodology used to compute the power figures, we now explain the method used to obtain energy measurements reported in the below tables. Figure 28 shows the circuit used as a reference for perform power measurements. For short, we went to measure the ΔV voltage difference drop across the shunt resistor R.

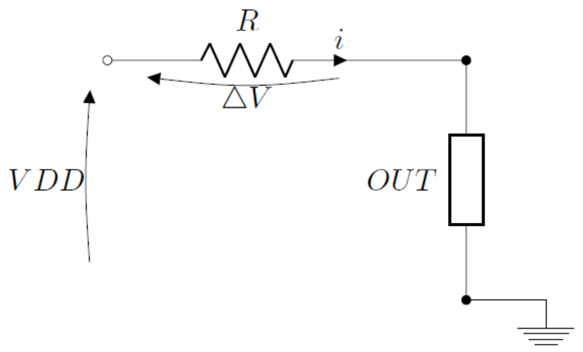


FIGURE 28: REFERENCE CIRCUIT USED FOR POWER MEASURES

The *Saleae Logic* software[32] was used to make the measurements through which it is possible to sample the measured signal, thus obtaining the $\Delta V(n)$ signal at the discrete time $\Delta V(t)$. Thanks to the sampled ΔV we can calculate the instantaneous power as follows:

$$P_i = \frac{VDD * \Delta V_i - \Delta V_i^2}{R}$$

where ΔV_i represents the voltage drop on the shunt resistor R of the *i-th* sample. Thus, thee energy consumed could be derived as follows:

$$E = T_s * \sum_{i=0}^{N} P_i$$

where T_s is the sampling period, N are the samples considered and P_i is the instantaneous power.

TABLE 3. ENERGY AND POWER CONSUMPTION

Energy and power figures	*mW*	*mJ*
ESN model (training)	6.285101	2.759411
AR Model (training)	8.925831	0.615811
ESN model (prediction)	9.123852	0.045838
AR model (training)	7.128756	0.035815
getSensors() (data acquisition)	7.368875	0.212501

[32] Saleae Support – Software Download - https://support.saleae.com/logic-software

The 1st row in Table 3 is related to the training of the ESN model for the accelerometer sensor, the 2nd row is about the measurement relating to the training of the AR reference model for temperature, pressure and humidity sensors, the 3rd row is about the ESN model in prediction mode, one for each of the three accelerometer axes, the 4th row is about the reference AR model, one for each temperature, humidity, pressure values. Finally, the last row shows the power consumption for acquiring a sample from the sensor shield. This latter reference figure is computed from accelerometer readings, sampled at F_s= 6.125 MHz.
We conclude with a final consideration about energy consumption through an example. Suppose we consider a test window of N=100 samples where the ESN model is in its operational phase (i.e., prediction mode) and the Change Detection Test (CDT) stage is analyzing the residuals. For n =1,2,…,100 the residual is calculated as $r(n) = y^{\wedge}(n) - y(n)$, denotes $y^{\wedge}(n)$ the prediction at the instant *n* of the real value $y(n)$. From an energy perspective this translates as:

$$E_r = \left(E_{getSensors()} + E_{prediction}\right) * N$$

where E_r is the energy consumed for N sensor data acquisitions and N model predictions. It is interesting to note that the analytics predictive part of the processing consumes around ~124% of the energy spent for just acquiring the data, and that the more complex ESN model translate into almost equivalent figures vs the simpler and more limited reference AR models, thus a quite good results.

4.4.3. DEPLOYMENT SCOPE

This algorithm has been characterized and tested both in a laboratory environment through dedicated tests, but also in real environments mapping it into the complete binary firmware mapped on STM32 MCU units board. The actual implementation, fully tested, has been released as a specific SW artefact to be integrated into final demonstrator during WP5 – task 5.6 activities. Final fine-tuning and adaptions will be made on a real deployment demo setup as part of T5.6 activities where it will be reported incrementally into D5.6 – “Demonstration and validation of IHES Generic IoT (Cycle 1)” and D5.10 – “Demonstration and validation of IHES Generic IoT (Cycle 2)” deliverables.

4.5. Change Detection Test

4.5.1. ALGORITHM DESCRIPTION

This algorithm is intended to identify changes in data dynamics detected by sensors. To do this, residuals computed by model prediction stage are used: they are defined as the difference between the value detected by the sensor and that predicted by the learned model. When the model approximates residuals well, they can be modeled as an i.i.d. (independent and identically distributed) and it is therefore possible to use the algorithms that are called Change Detection Test (i.e., CDT as described in [32]). Only non-CDTs will be considered in this work parametric or that once again, do not require any information a prior to the distribution of the variable taken into consideration. In particular, two CDTs have been selected and implemented in this phase:

- Majority voting threshold-based CDT [32]: this method uses a threshold (calculated using any number of samples) and, when, during the operating phase, *n* residuals exceed the threshold in a given time window, that moment is considered a change.
- ICI-based CDT [33]: this solution calculates the average of the values using non-overlapping windows. These averages values are then used to calculate, through the rule of the confidence intervals, possible changes.

Both these methods have been selected for their limited computational complexity, which, on the other hand, leads in both cases to have an increasing percentage of false positives. For this reason, a further validation phase has been introduced with the aim of determine which changes are associated to real observed anomalies and which are not. The algorithm used is described in section 4.6.

4.5.2. VALIDATION AND TESTING

For the CDT algorithm characterization, the focus is about the false positives detection minimization. A set of specific tests has been done to characterize this aspect of the algorithm using as accelerometer sensor as source of input data. The test has been done by defining a dataset of several acquired temporal sequences of 70 seconds each acquired at 5Hz sample rate. A synthetic noise has been added to each given sequence to simulate a change. Three main noise models were considered:

- Random noise (device fault)
- Ramp noise
- Stuck-at (constant signal)

Given this test environment, using each of these sequences an instance was calculated of the ESN model and the thresholds for CDT have been calculated. Then the model was used in operating mode on the remaining part of signal up at the time a change was detected. Now let us define some parameters with which we will then describe, once the change is detected at time T_{ref}, what are the possible situations:

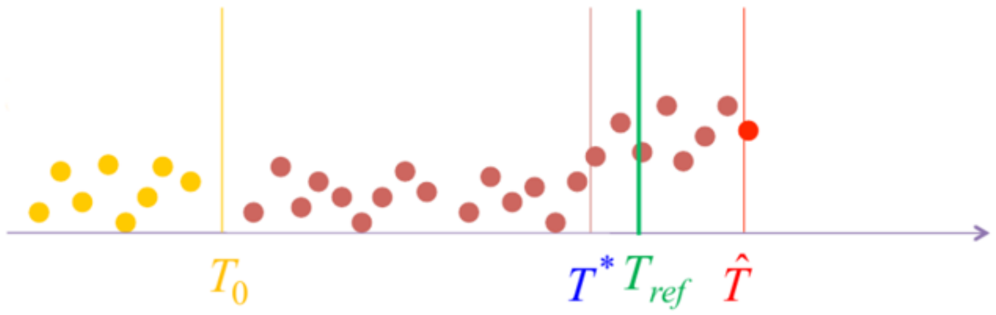


FIGURE 29: FALSE POSITIVE DETECTION METRIC

With reference to Figure 29 above, we define:

• T_0: the instant of time in which the experiment begins (the model parameters and the nominal state has been calculated);
•T^*: time when the noise signal is introduced, and the change begins;
• T_{ref}: time when the change is detected by the model
just trained;
•$\hat{T}$: the instant of time in which the experiment ends.

Based on these definitions, three cases can be possible:

• Detection of a false positive: $T_0 < T_{ref} < T^*$
• Correct detection: $T^* < T_{ref} < \hat{T}$
• Detection of a false negative: $T_{ref} > T^*$

In case of correct detection, an additional value is defined, the detection delay (*dd*) defined as:

- $dd = T_{now} - T^*$

where T_{now} is the current reference time after the execution of the CPM algorithm.
Various experiments have been carried out, and their used parameters will be reported together with the results. In particular, for both tables presented below the following test configuration has been defined:

- Training samples: 100
- Test sequence length: 350 samples (70 seconds @ 5Hz sampling)
- Total number of experiments done: 200

The only variable parameter in the two tables is *p*. It is nothing but a factor multiplicative applied to the threshold calculated and used during the CDT phase: Table 4 uses *p = 2.5*, Table 5, instead, *p = 5.0*. This means that if the threshold computed by an ESN instance (trained on the first 100 samples of a sequence) turns out to be 10, the change will be detected using a threshold equal to *25* in the experiments reported in the first table and equal to *50*, for those shown in the second.

TABLE 4: AVERAGE RESULTS ON 200 EXPERIMENTS WITH *P=2.5*

	Fault			Incremental			Stuck-at		
	PFR	FNR	dd	PFR	FNR	dd	PFR	FNR	dd
AR	100%	-	-	100%	-	-	100%	-	-
ESN	5%	-	40,7	0.8%	-	97.3	13%	29%	21.7

TABLE 5: AVERAGE RESULTS ON 200 EXPERIMENTS WITH *P=5.0*

	Fault			Incremental			Stuck-at		
	PFR	FNR	dd	PFR	FNR	dd	PFR	FNR	dd
AR	81%	-	49.5	70%	-	50.5	63%	-	50.5
ESN	-	-	40,7	-	-	49.5	-	32%	21.7

Legenda: PFR= false positive rate; FNR = false negative rate; dd = detection delay

4.5.3. DEPLOYMENT SCOPE

Similarly, to the time series prediction algorithm presented in section 4.4.1, the CDT algorithm will also be integrated into the final UC3 demonstrator where final fine-tuning and adaptions will be made on a real deployment demo setup as part of Task 5.6 activities: this part of the development will be reflected incrementally into D5.6 – "Demonstration and validation of IHES Generic IoT (Cycle 1)" and D5.10 – "Demonstration and validation of IHES Generic IoT (Cycle 2)" deliverables.

4.6. Change Point Method

4.6.1. ALGORITHM DESCRIPTION

Once one of the CDT adopted has confirmed a change, the validation stage is activated. The goal of the Change Point Method stage (CPM) is to reduce the occurrence of false positive detections and to provide an estimate of the time instant when change started, while trying not to introduce further delays in the change detection. To achieve this goal, a powerful (from the statistical point of view) and more complex (hence more computational

demanding) statistical technique has been developed that does not require to operate in an on-line sequential manner.
In order to achieve this goal, the validation stage takes place through hypothesis tests named Change-Point Methods (or CPM described in [34]): these methods are statistical hypothesis tests operating on a fixed data sequence.
These statistical tests verify whether, within a sequence of values, there is a so-called Change-Point: a point where the data generation function changes its probability density. If this point is found inside the sequence, the change is confirmed and that turns out to be the point where actually the signal has started to appear as abnormal.
CPM stage operates as follows: for each possible partition of the fixed data sequence, a suitable test statistic is computed. When the maximum value of such a statistic overcomes a threshold computed on a given confidence level as described in [35] then a change-point is present at that confidence level and the index associated with the maximum value represents (suitably transformed in the temporal domain) the estimated *Tau* value. The CPM algorithm is used to enable self-adaptation by activating suitable reaction mechanisms (i.e., retraining the IHES sensing node).
The firmware implementation of these hypothesis tests is carried out using a CPM test, specifically based on the Mann-Whitney test [36], thus working on the mean of the residual values to verify the presence of the Change-Point and when it precisely began at *Tau* instant.
Formally, the Mann-Whitney CPM operates as follows. Given a data sequence of length n:

$$\mathscr{X} = \{x(t), t = 1, \ldots, n\}$$

all the possible partitions

$$\mathscr{A}_\tau = \{x(t),\ t = 1, \ldots, \tau\},$$
$$\mathscr{B}_\tau = \{x(t),\ t = \tau + 1, \ldots, n\}$$

are computed and the following test statistic is computed

$$D_\tau = \sqrt{\frac{\tau(n-\tau)}{n}}\,\frac{\bar{\mathscr{A}}_\tau - \bar{\mathscr{B}}_\tau}{S_\tau}$$

where $\bar{\mathscr{A}}_\tau$ and $\bar{\mathscr{B}}_\tau$ denote the sample means evaluated on $\mathscr{A}_\tau$ and $\mathscr{B}_\tau$ respectively and S_τ is the pooled sample variance evaluated on $\mathscr{A}_\tau$ and $\mathscr{B}_\tau$. The threshold $h_{n,\alpha}$ for the statistic D is provided by the Student *t* distribution with n - 2 degrees of freedom.
In practice,

$$\begin{cases} \text{The estimated change-point in } \mathscr{X} \text{ is } \boldsymbol{\tau} & \text{if } D_\tau \geq h_{n,\alpha} \\ \text{No change-point identified in } \mathscr{X}, & \text{if } D_\tau < h_{n,\alpha} \end{cases}$$

4.6.2. VALIDATION AND TESTING

An example of how the Change-Point works, is given in Figure 30. There the length of the sequence is 500 samples. The (unknown) change point is at sample 350. Figure 30.a shows the distribution of the samples (in our case the residuals). Figure 30.b shows the corresponding value of the test statistic Ds for all the possible partitions. The maximum value that is in TM is above the threshold $h_{n,\alpha}$(the horizontal dotted line).
In the Figure 30, 200 represents just an example of computation of the test statistic on the two partitions [1…,200] and [201…,500], while the true change point is at 350.

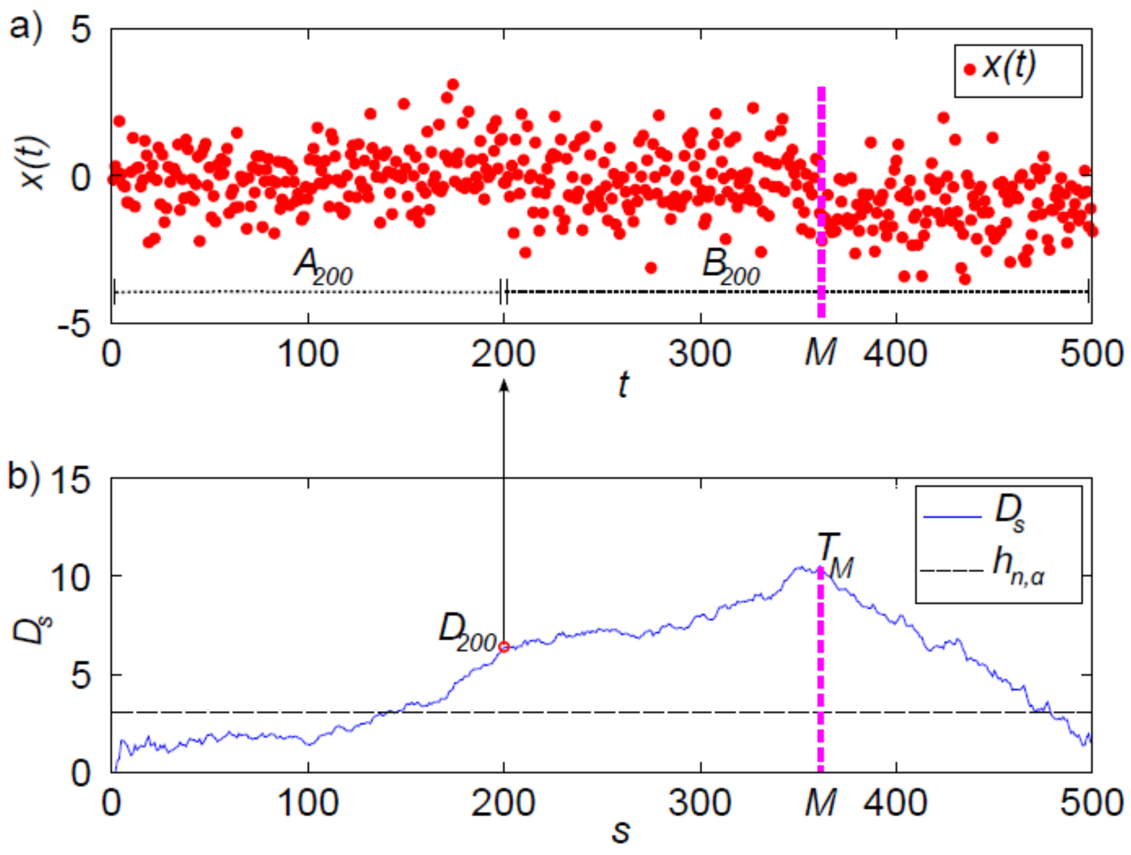


FIGURE 30: AN EXAMPLE OF CPM USAGE[33]

4.6.3. DEPLOYMENT SCOPE

Similarly to the time series prediction algorithm presented in section 4.4.1, and the CDT algorithm described in section 4.5.1, also the CPM algorithm will be integrated into final UC3 demonstrator where final fine-tuning and adaptions will be made on a real deployment demo setup as part of Task 5.6 activities: this part of the development will be reflected incrementally into D5.6 – "Demonstration and validation of IHES Generic IoT (Cycle 1)" and D5.10 – "Demonstration and validation of IHES Generic IoT (Cycle 2)" deliverables.

4.7. Oil Leakage Detector

4.7.1. SYSTEM & ALGORITHM DESCRIPTION

In Use Case 1 the problem of a possible oil leakage inside a wind turbine arises. The later this oil leak is discovered, the higher the resulting damage. The use of a camera inside the turbine is a cost-effective solution to detect the oil leakage at an early stage. An ML model is used for this purpose, which is operated on the edge devices installed on the wind turbines and performs image analysis. This image classification process has to be specially trained for oil leakage. The trained model can then be used for all turbines for permanent analysis during operation and is obtained from a central location (data center). In addition, the model will be further

[33] Here, the length of the sequence is 500 samples. The original change point is at sample 350. Figure 30.a shows the distribution of samples. Figure 30.b shows the corresponding test statistic.

refined over time. This is necessary because the turbines have different designs and sizes, which means that the camera perspective can vary. This results in variances in the image motifs to be analyzed.

Since the training data never leaves the client, bandwidth is saved if the size is appropriate. In order to prevent that conclusions from the new model can be drawn from the private training data through reverse engineering [37], additional approaches are available. Differential Privacy mathematically ensures that different types of statistical analysis do not affect privacy. Thus, statistical analyses of a data set must not allow any conclusions to be drawn about an individual. Secure Aggregation uses encryption to make the individual updates of the clients on the FL server uninspectable.

This repeated training for optimization should also take place on the Edge devices using the Federated Learning approach. To do this, the Edge devices retrieve the current model from a data center, train it further using their local training data, which is classified by a technician, and then return an update. The global model is then updated at the data center using the update received and later communicated to all Edge devices.

This FL approach has two advantages. First, the training data set on the Edge device does not need to be transferred to the data center. The wind turbines are often located in rural areas or at sea and therefore have limited bandwidth. The transmission of large amounts of data should therefore be avoided. On the other hand, the Federated Learning approach has the advantage that the training data does not leave the Edge device. Any trade secrets can therefore be kept secret.

The following describes a system for distributed training of a screen classification. It consists of several physically separate nodes. There is always a central node, called the FL server. This device has the component Orchestrator, which is responsible for coordinating the entire training process. In addition, the hyper-parameters required for the training process can also be defined there. The number of other nodes can be unlimited. The devices are called clients and have the Worker component, which is responsible for training. Figure 31 below shows a detailed overview of the system. In this case, three Clients are used for training. There must be a connection between each client and the FL server for data exchange.

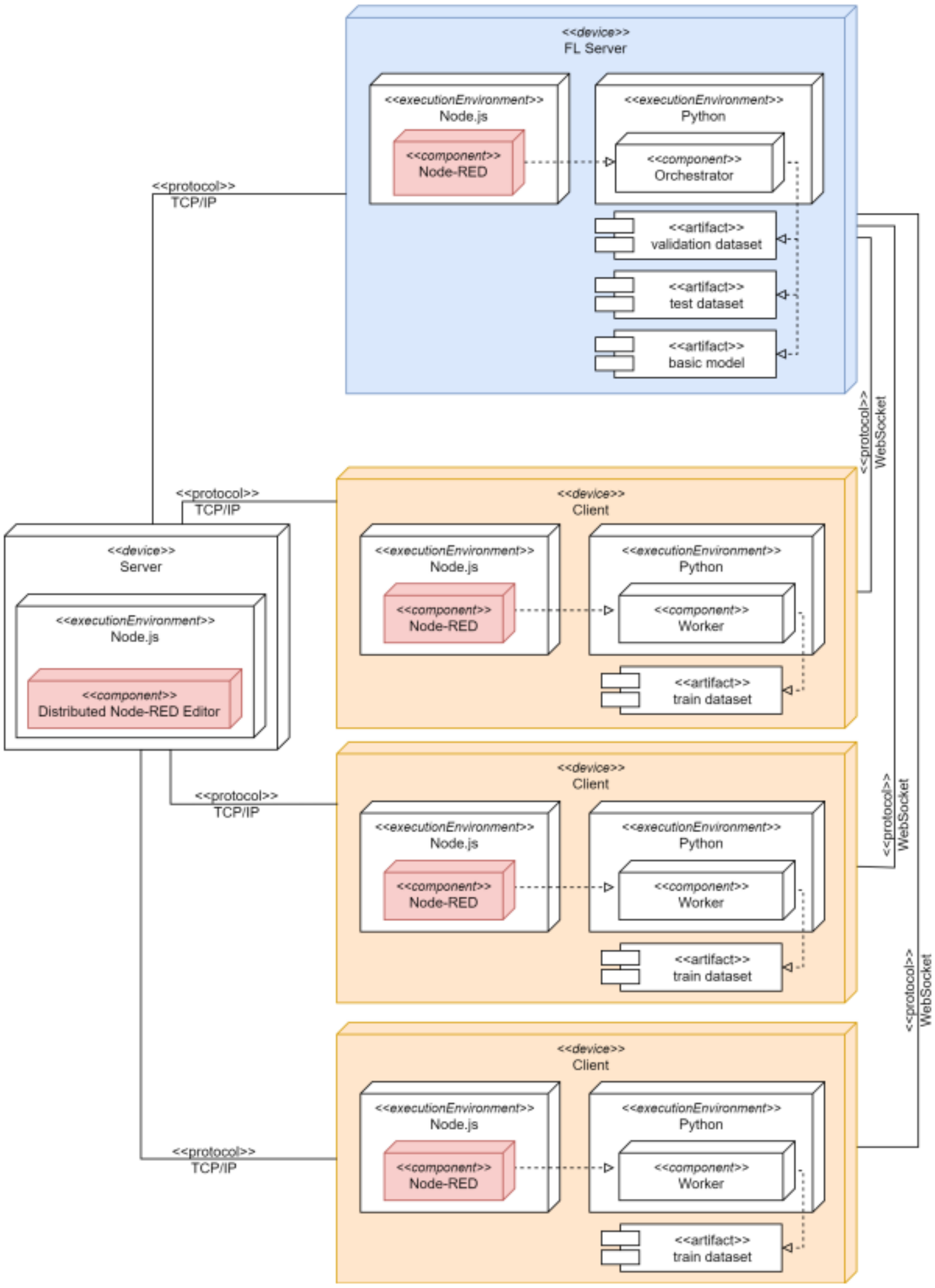


FIGURE 31: NODE-RED SYSTEM DESIGN

The actual training process is executed on the clients. Therefore, all clients must have their own training data. The FL server has no access to this data at any time.
The system is based on the Python library PySyft. This library was specially designed for secure and private deep learning. The PyTorch machine learning library is used as the basis.
Each node requires a Python runtime environment. The Python Component Orchestrator located on the FL server communicates with the Python Component Worker located on the clients via the WebSocket protocol.

This TCP-based network protocol enables a bidirectional connection. Data is sent from Orchestrator to the workers and vice versa. The connection is established once and then remains active throughout the entire training process.

The Python Component Worker has two primary functions One is to reference the local training record. This requires the file path where the training data is located and a key. The names of the classes in which the images are to be classified are defined by the directory names. All images within a directory are therefore automatically assigned to the class with the directory name. The directory name therefore also serves as a label. Before the images can be used for training, they must be transformed. All images must have the same resolution for training the neural network, because the number of pixels results in the number of input features (input parameters for the neural network) and this number is fixed. To obtain a greater variance in the training data set, a section of the image is first created with a random size and then scaled to a random aspect ratio. There are limits for both values so that the subject can still be recognized and the image does not become unusable. Then the photo is cropped to the specified size so that the resolution of all images is the same. Furthermore, a probability is used to decide whether the photo should be rotated horizontally to further increase the diversification.

A worker can theoretically provide several data sets. Therefore, each local data record must be referenced by a key. All workers who participate in a training process should use the same key for the local data record. The orchestrator then uses this key to specify which data record is to be used for training without having access to the data. If the workers assign different keys, each key would have to be stored with the orchestrator, referenced by the corresponding worker.

On the other hand, a TCP port is opened for the WebSocket connection and waits for an incoming connection from the Orchestrator. For this function PySyft offers the class WebsocketServerWorker.

The FL server has the Python Component Orchestrator and has access to the validation and test data set. The corresponding file path must be known to the Python component. As with the clients, the directory name of the data records is responsible for the class names on the FL server. The directory name therefore serves as a label again and the images of the corresponding class must be located in the respective directory. The images must also be transformed. However, random cutting or scaling is not necessary for these data sets, as they are not used for training. In the Orchestrator, the images are only cut in the middle to the defined resolution, so that the images in the validation and test data set have exactly the same resolution as in the training data sets.

The orchestrator determines the structure of the neural network. However, the network is not defined from scratch, but the transfer learning approach is followed. The basis for the neural network is therefore an already proven model. Since the number of classes can differ from the available model, the output features of the last layer have to be adjusted to the corresponding number of classes. The number of input features must also match. The proven model has already been trained with a large number of images with a certain resolution. If the available images have a different resolution, the input features must also be adjusted. Alternatively, a self-trained model can be passed as the base model. For example, the saved model of a previous call of the Orchestrator component can be used as base model for the next call. If no model is passed, the Convolutional Neural Network MobileNetV2[34] was used as the initial model. The architecture of this model has been specially adapted for resource-constrained environments. MobileNetV2 can distinguish between 1000 different classes by default. Alternatively, a different model can be used as the initial model for classification. The models can be obtained via Torchvision[35] and can thus be downloaded directly in the Python script.

Either only the model architecture can be loaded or a model pre-trained with ImageNet[36]. The ImageNet database contains millions of images and is often used in research projects. All models have a different number of parameters that need to be trained. The appropriate model can be selected depending on the application and available computing power.

Furthermore, the orchestrator establishes the web socket connection to the workers. For this purpose, the corresponding network parameters (IP address and port) of the workers are required. In addition, each worker has an ID to enable meaningful log output. The workers receive new training configurations for each training round. These include the model and the error function, which must be made serializable for the transfer. PyTorch

[34] https://arxiv.org/abs/1704.04861
[35] https://pytorch.org/docs/stable/torchvision/models.html
[36] http://www.image-net.org/

provides the necessary functions with TorchScript[37]. The other parameters are simple numerical values which can be easily serialized.

It should be possible to create, coordinate and start the training process via a central node with a graphical interface. This has the advantage that both Worker and Orchestrator do not have to be activated manually with the appropriate parameters. In addition, the visualization of the training process increases the clarity of the system. The graphical development tool Node-RED, which is widely used in the IoT environment, is used for this implementation. Its use brings the following additional advantage: The Node-RED ecosystem allows the application to be flexibly expanded for future use cases.

All devices participating in the training process must have Node-RED installed and started as a process. To distribute Node-RED processes to different devices, the Distributed Node-RED Editor is required. This has been specially designed for this and is installed on a separate server. Figure 32 shows the structure of the system including Node-RED.

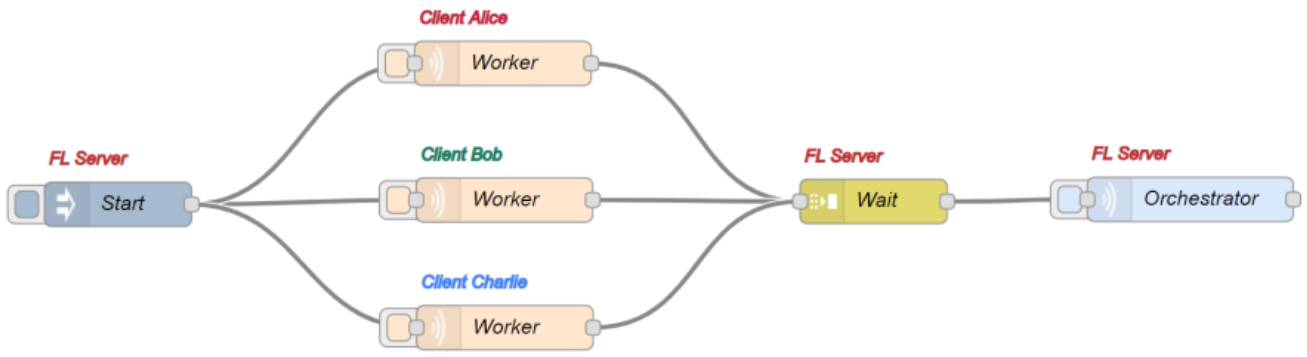
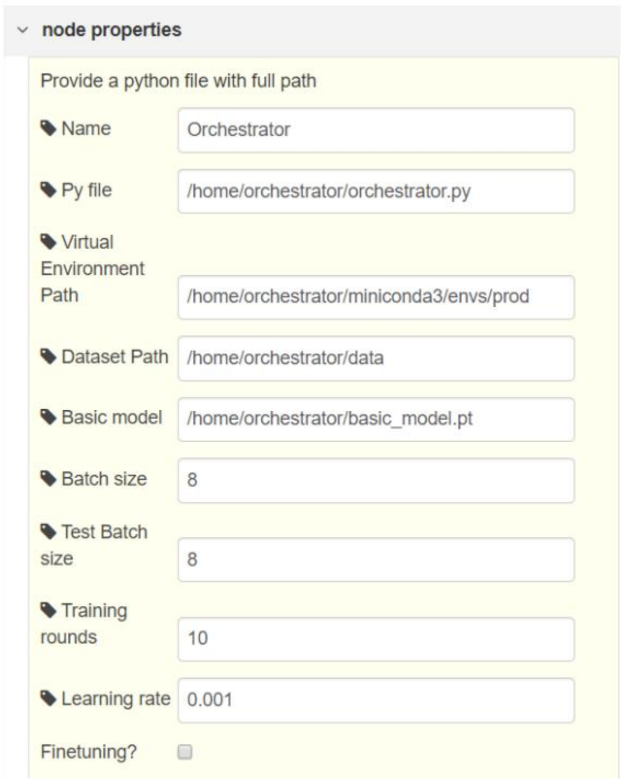


FIGURE 32: INSTANTIATED NODE-RED FLOW

For the Federated Learning process special nodes for Node-RED were created. There is an orchestrator node and a worker node. These must be installed on all devices for the training process. Both were not developed from scratch, but are based on the already available node node-red-contrib-pythonshell[38]. This can already start a Python script from node-RED. The input of the node is passed as an argument to the Python script and the output of the Python script is sent as output of the node. The node has been adapted so that the parameters required for the training process can be configured in Node-RED. The path to the respective Python script must also be specified. The scripts can therefore be located on any device at any file path. It is also possible to use a virtual Python environment. Figure 33 shows the Node-RED interface to configure the two nodes.

[37] https://pytorch.org/docs/jit.html

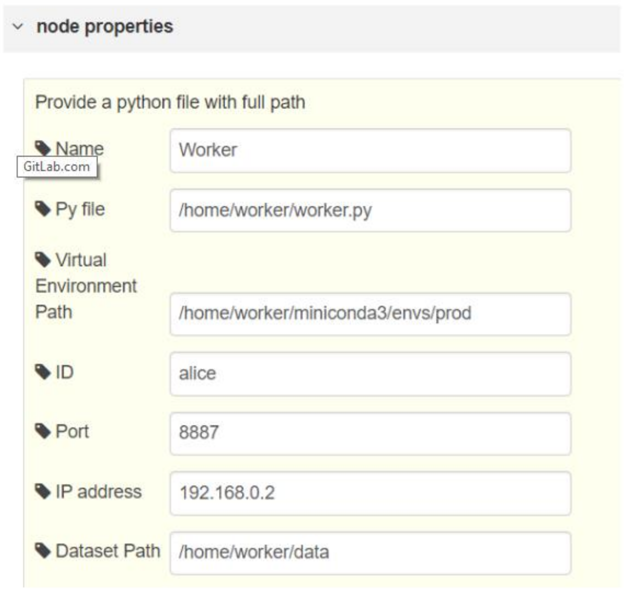
[38] https://github.com/namgk/node-red-contrib-pythonshell



node properties

Provide a python file with full path

Name	Orchestrator
Py file	/home/orchestrator/orchestrator.py
Virtual Environment Path	/home/orchestrator/miniconda3/envs/prod
Dataset Path	/home/orchestrator/data
Basic model	/home/orchestrator/basic_model.pt
Batch size	8
Test Batch size	8
Training rounds	10
Learning rate	0.001
Finetuning?	☐



node properties

Provide a python file with full path

Name	Worker
Py file	/home/worker/worker.py
Virtual Environment Path	/home/worker/miniconda3/envs/prod
ID	alice
Port	8887
IP address	192.168.0.2
Dataset Path	/home/worker/data

FIGURE 33: CONFIGURATION DIALOGS FOR ORCHESTRATOR AND WORKER NODE

4.7.2. DEPLOYMENT SCOPE

The system described above is now to be used for oil detection in wind turbines. A wind farm consists of a collection of several wind turbines (WTGs). For administration purposes, the individual WTGs are connected to a central location called a data center.

Each wind turbine has its own network in which various devices such as industrial PCs (IPC) and programmable logic controllers (PLC) are located. A wide variety of sensors, such as temperature sensors or accelerometers, are already connected to the IPCs. The PLC is responsible for controlling the rotors.

For oil detection, a camera is now also to be connected to one of these industrial PCs. There are various areas where oil can tend to leak. One area is particularly critical, which is why the camera should be aligned to this area. Since it is dark inside the turbine, an infrared camera with infrared emitter is used. This is able to take recognizable pictures even in complete darkness. The LEDs built into the infrared emitter send a light into the infrared range which is invisible to humans. The infrared camera can take a picture in gray scales by this radiation.

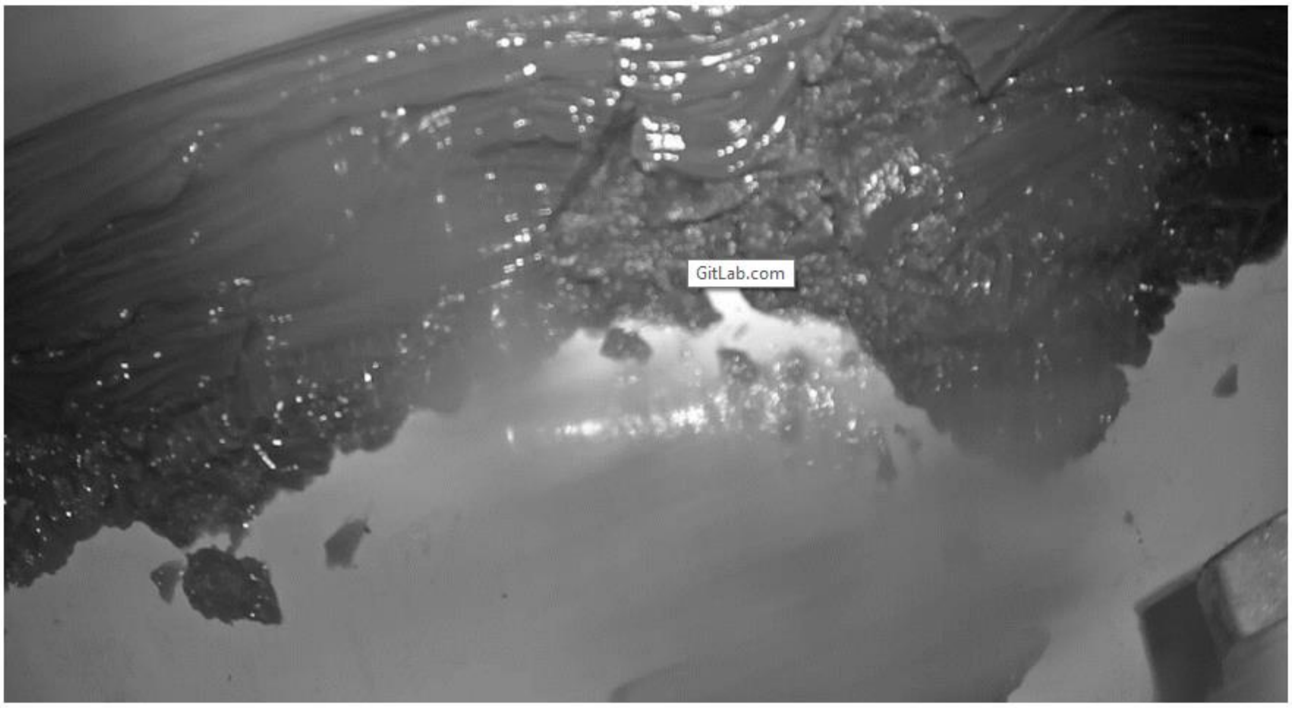


FIGURE 34: OIL LEAKAGE IN WIND TURBINE

Figure 34 above shows a picture of the oil leakage in a wind turbine, which was taken with an infrared camera. For the image analysis a basic model is used, which will be improved over time. Since the oil leakage is rather slow, one or two images per day are sufficient. The photos are stored on one of the industrial PCs with a time stamp. A technician comes approximately every 90 days to carry out service work. During this service work, the technician also determines the labels of the images. If no oil is visible on the wind turbine, all images are given the label no_oil. In this case the technician does not need to look at the images at all. If oil has leaked, it is sufficient to determine the time of leakage from the pictures. For this purpose, the images can be viewed backwards using the time stamp. For the first picture without oil, the labels can already be determined. From this point on, all images before that receive the label no_oil and all images after that the label oil. New training records are then created from the images. It is important that the approximate number of images with and without oil is the same in all training records. Otherwise, the neural network tends to be more of a class. However, one can assume that the number of images without oil is significantly higher. This can be considered in the training process, because not all the wind turbines have to participate in the training. So an imbalance in the data sets must be avoided. As soon as several wind turbines have a training data set, the training process can be started. In this scenario the clients are the IPCs on the wind turbines. The Worker component must be started on these IPCs. The Orchestrator component runs on the FL server in the data center. The individual wind turbines are then connected to a central data center of the wind park. This connection is required for data transfer in the training process.

As there is currently too little actual imagery data from a real wind turbine available. For this reason, the functionality of the system will be demonstrated using a test installation. The resulting images should be as similar as possible to real wind turbine images. The test turbine was first designed as a 3D model with a CAD program. Figure 35 shows a rendering of the 3D model. The arrow points to the location relevant to the photos.

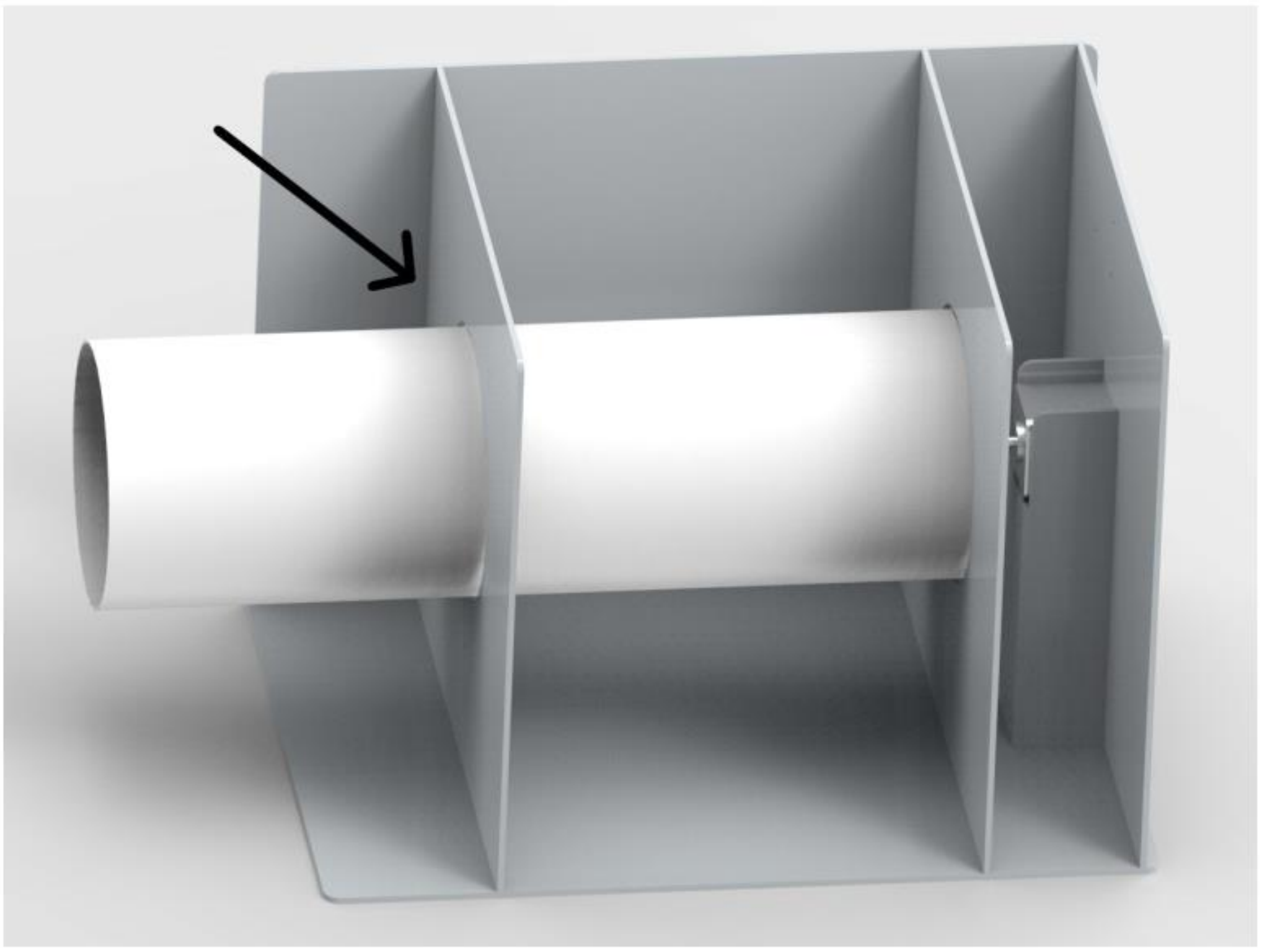


FIGURE 35: 3D MODEL OF TEST FACILITY

The test system was then made from several acrylic glass sheets. These were cut with a laser cutter according to the specifications of the 3D model. Afterwards, the individual sheets could be inserted into each other through suitable cut-outs. A PVC pipe was used as a moving component to ensure that the oil was distributed evenly. This creates the characteristic smear marks on the pictures. For the rotation a motor was attached, which is placed on a platform and fixed with the pipe. The motor is controlled by an Arduino.

In order to achieve the highest possible comparability between the pictures of the test facility and the pictures of the real wind turbines, an infrared camera with infrared radiator is also used. This gives the images the same picture style as in the real application. Now the data set could be created. The arrow in Figure 35 points to the location to be photographed. First of all, the pictures were taken without oil. To make the work easier, however, short videos were taken instead of direct pictures. It was important to find as many different perspectives as possible. The rotation of the pipe also helped to obtain the greatest possible variance in the data. Next, the oil could be applied to the pipe at the point marked on Figure 35. In the pictures, it should look as if the oil is seeping step by step between the pipe and the plate. The amount of oil was gradually increased. Here too, it was important to change the camera perspective. The rotation of the pipe created the smear effect.

After the videos were created, they could be converted into pictures. The tool *FFMPEG* was used for this. An image was created for every second in the video. For the training process it is important that the number of images of all classes is equal. In total, about 600 images per class were generated for the test system data set. Figure 36 shows one image with and one without oil from the created data set.

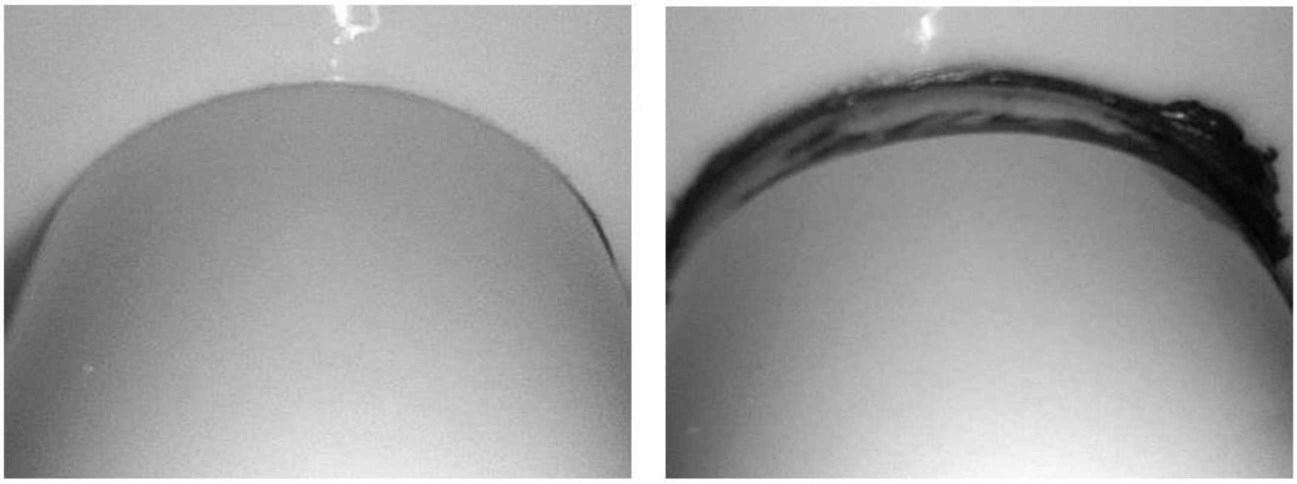


FIGURE 36. IMAGES OF THE TEST FACILITY DATA SET

Afterwards the images were divided into the different data sets. Altogether three workers are used for the training, each of whom has a training data record. In addition, there is the Orchestrator, which needs a validation and a test data set. The size was uniformly set to 200 images per data set. The images were removed from the entire data set in random order and added to a new data set. The remaining images were assigned to an evaluation dataset. These can be used for evaluation after the training process, i.e., for simulation of the running operation.

Now the data sets and the Python components could be distributed to the devices. To avoid possible side effects with other Python packages, virtual environments with *conda* were created on all devices and then the required packages were installed. For the training process, all devices must be on the same network. The devices were connected to a router, which also served as a DHCP server. In addition, Node-RED was installed on all clients and the FL Server and Distributed Node-RED on another server. After installation and configuration of the individual nodes, the training process could be distributed and started via Node-RED.

4.7.3. VALIDATION AND TESTING

In the following the two metrics error and accuracy are calculated and explained for the data set of the test facility. With classical ML algorithms there is only one model, which is improved epoch by epoch. Here the two metrics for a training and a validation data set are calculated for each epoch and compared at the end of the training. Since there are several training data sets in Federated Learning, you also get several values for error and accuracy. Each worker calculates these for his or her course data record after each epoch. In addition, the metrics are calculated after each training round for the validation data record using the aggregated model (federated model). A training round is completed when all workers have completed an epoch.

In the Orchestrator component there is the flag EVAL_ON_WORKER. Only if this flag is set will the evaluation be executed after the training. The reason for this is that the calculation of the metrics leads to an increasing runtime of the training process.

For the evaluation, the training was distributed among the three workers Alice, Bob, and Charlie. To minimize the runtime for this test case, all workers and the Orchestrator were started on the same device (and operating system). The data is transferred via the WebSocket protocol, but is not really sent over a network. Only the loopback interface was used for the transmission, since it is not important for the results of error and accuracy. A total of 5 training rounds were executed with a learning rate of 0.001. These two hyper-parameters have a decisive influence on the training. The difficulty is to find the optimal values for the hyper-parameters. The values form a good average value and were therefore used for the start.

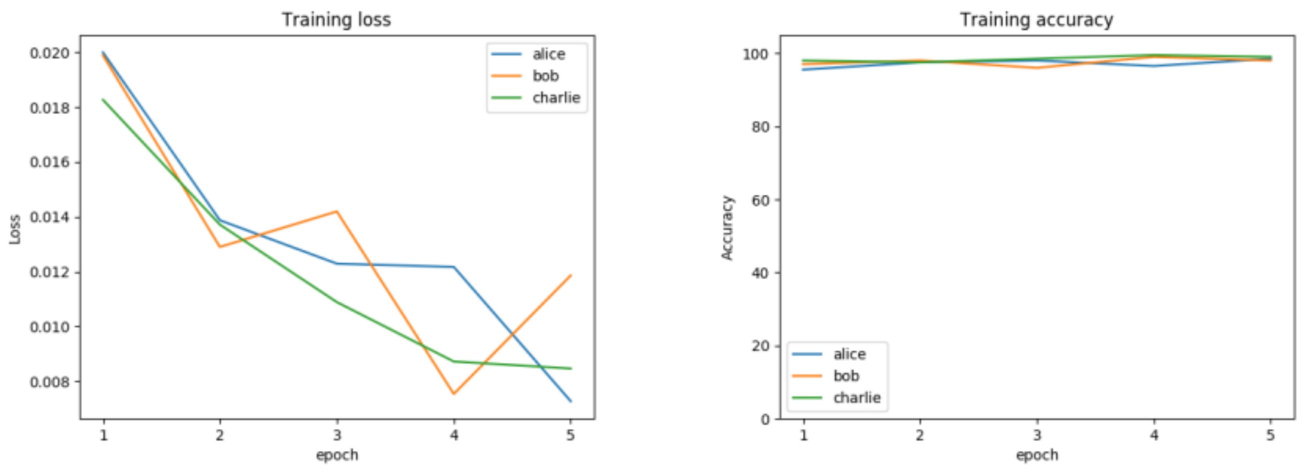


FIGURE 37. ERROR AND ACCURACY OF WORKER MODELS

Figure 37 above shows errors and accuracy of the training records of the three workers. Please note that each worker has his own model and training data set. Since the pre-trained model MobileNetV2 was used, the error after the first epoch is already very low and the accuracy is very high. The accuracy is between 98% and 99% for all workers after the training process. The excellent results of the training data sets are reflected in the results of the validation data set (see Figure 38). For this data set, the aggregated model (federated model) was used for the calculation.

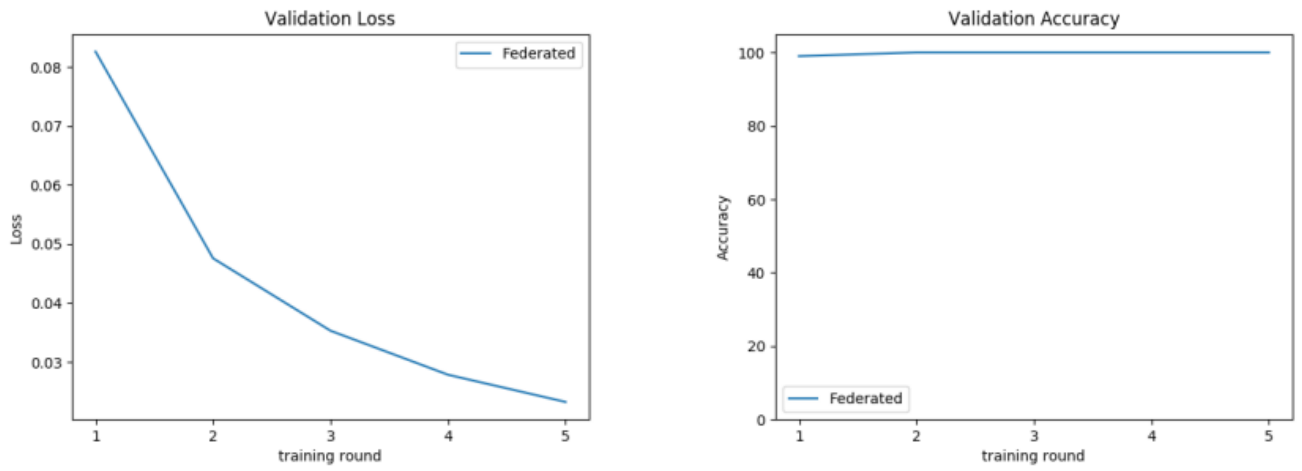


FIGURE 38: VALIDATION LOSS AND ACCURACY OF AGGREGATED MODEL AT ORCHESTRATOR

After the training phase has completed, the WireShark recording could be analyzed. By means of the packet length and the timestamp it can be recognized from when the worker starts training. After the connection has been established, the training configurations are first sent from the orchestrator to the worker. During this time the worker sends only ACK packets. The length of the consecutive packets sent by the Orchestrator is relatively long and the time intervals are minimal. This is followed by a longer time interval in which the worker trains. After training, the worker returns the results in longer packets. At this point, the orchestrator only signals the receipt of these packets with ACK packets. Now the used bandwidth is measured in both directions. For this purpose, the messages are additionally filtered by the source IP address. First the amount of data transferred from Orchestrator to worker is determined. The specified source IP address in the filter is thus the IP address of Orchestrator. The recording was then exported as a CSV file for the calculation and imported into Excel. There the lengths of the individual packets could be added together. Afterwards the same procedure was performed with the Worker IP address as source IP address. The Table 6 below shows the result. To make sure that the results are reliable, the test was run several times in a row.

TABLE 6: TRANSMITTED DATA BETWEEN ORCHESTRATOR AND WORKER

Orchestrator to Worker	8,7 MB	18,6 MB
Worker for Orchestrator	8,7 MB	18,6 MB

The amount of data transferred is therefore approximately the same in both directions. The largest part of the transmission is determined by the size of the model. The reason for the significant difference between the amount of transmission and the size of the model is the PySyft implementation. The model is converted to a binary serialization format using Python MessagePack. This serialized message is then converted to a hexadecimal representation using the binascii library. The method hexlify() is used for this. This method converts each data byte into the corresponding two-digit hexadecimal representation. The returned byte object is therefore twice as long as the length of the original data. On receiving, the data is converted back to the original format using the unhexlify() method. The reason for the considerably larger amount of data during transmission is therefore the conversion to the hexadecimal representation.
In addition, there is significant potential for improvement in the PySyft implementation. The implementation allows both the feature extract and the fine tuning mode for transfer learning. By default, the latter is used because the pre-trained MobileNetV2 already delivers good results. Therefore, only the weights of the last shift are optimized during training. It would be more efficient to transfer only this layer and not the whole model. A possible solution is to split the model into two models. The first model (frozen_model) consists of all so-called frozen layer These are the layers whose parameters cannot be changed. In this case, these are all but the last layer. This model only needs to be transferred to the workers once. The second model (trainable_model) has the trainable layers, in this case only the last layer.
The differences between MobileNetV2 and frozen_model are only in the kilobyte range and can therefore be neglected. The file size of the trainable_model is much smaller with 11 KB. If only this model were to be sent from the second training round onwards, an enormous amount of bandwidth could be saved.

In the following, it will be shown how well the trained model works during operation. The model was developed for this purpose analogous to the first run (5 training rounds and 0.001 fixed learning rate) and saved at the end. Furthermore, a separate Python program was created for classification during operation. First of all, the images of the evaluation data set are loaded into this program. This data set contains 187 images, which are unknown for the trained model. Then 12 images are propagated through the neural network in a loop. With the help of the library Matplotlib these images are graphically displayed together with the probabilities predicted by the model. After the user has closed the graphic, the next loop run is started in the program and thus the evaluation of the next 12 images is started. Figure 39 below shows a screenshot of the images with their probabilities.

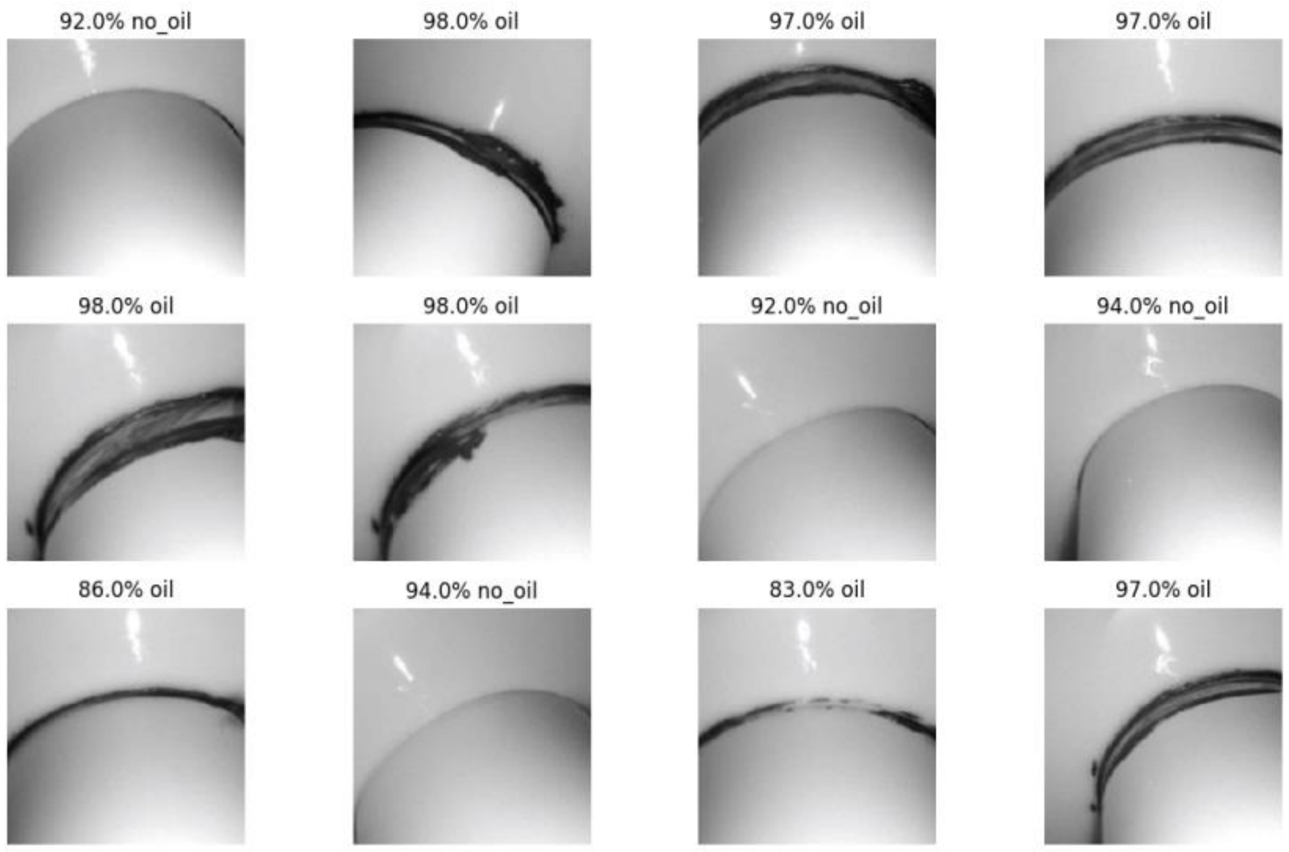


FIGURE 39: CLASSIFICATION OF IMAGES

The model has correctly classified all images. On closer analysis, it is noticeable that with an increasing amount of oil, the probabilities also increase. This means that the more oil there is, the more reliable the algorithm is. This is a further indication that the algorithm is working correctly. For example, in the second image in the first row, which shows a lot of oil, the algorithm is very secure at 98%. On the other hand, the third image in the third row has a comparatively low probability of 83%, which is due to the significantly lower oil content.

5. CONCLUSION

SEMIoTICS contributes to the architecture’s scalability by implementing specific local analytics algorithms at each layer of the SEMIoTICS architecture. When it comes to embedded devices in particular, this deliverable outlines an unfortunate trade-off followed by most IoT applications: devices are not taking advantage of their processing capabilities to avoid sending vast amounts of data to the cloud. Aside from privacy and scalability concerns, this practice spends valuable energy on the edge to communicate raw data to clouds.

Instead, SEMIoTICS proposes performing more computation on the edge, at a lower energy cost, and sending fewer bytes of information to the network and backend for further processing and event data aggregation, only when needed. In a nutshell, our approach ensures that devices can save energy overall, can protect the data’s privacy, and support massive scalability of the infrastructure by spreading these local analytics smart agents at the edge of the architecture. More to the point, considering the recent advances in AI/ML, this document provided a detailed description of the landscape for some ML tools and possible ways to design and implement more lightweight versions of dedicated algorithms for constrained devices. The range of possibilities for statistical and ML algorithms modelling have been presented in section 2.3.

Additionally, we complemented the view of the state-of-the-art techniques by describing the foundations for local analytics in embedded devices for SEMIoTICS in section 2. Moreover, the base for the local analytics functionality has been presented in several heterogeneous field of application: rule-based learning, time series prediction, time series clustering and federated learning. These approaches provide support for local embedded intelligence and local analytics within the project for the processing of generic time varying signals as well as video images. They are provided as working functional modules that have been identified as part of the cycle1 activities and developed during task 4.3 cycle2 activities. For some of them, an actual deploying of their functionalities has been already done on the selected target platform of interest, for others final characterization will be done during WP5 related activities. Thus, during cycle 2 activities, we have implemented the majority of all required local analytics functionalities that will be integrate in the SEMIoTICS specific use cases according to the specific integration plans of the different demonstrators within the WP5 task 5.4 to task 5.6 activities. Consequently, during WP5 and in particular during the integration of the three main SEMIoTICS demonstrators, final fine-tuning and integrated testing of the developed components and software artefacts are planned, and where all the main outcomes related to impacted KPIs will be reported as well. As part of this integration phase, some further refinement / enhancement could be additionally included, depending on emerging needs / issues reported from the specific use cases. These algorithms are likely to be integrated into final demonstrators as partially anticipated in Task 4.6 *“Implementation of SEMIoTICS backend APIs”*, where the interfaces to interface with each local embedded analytics algorithm have been defined from all the related requirements.

6. REFERENCES

[1] A. Kumar, S. Goyal and M. Varma, "Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things," in *International Conference on Machine Learning*, pp195-1944, 2017.

[2] C. Gupta, A. S. Suggala, A. Goyal, H. V. Simhadri, B. Paranjape, A. Kumar, S. Goyal, R. Udupa, M. Varma and P. Jain, "ProtoNN: Compressed and Accurate kNN for Resource-scarce Devices," in *International Conference on Machine Learning pp1331–1340*, 2017.

[3] M. Denil, B. Shakibi, L. Dinh and N. D. Freitas, "Predicting parameters in deep learning," *Advances in neural information processing systems,* p. 2148–2156, 2013.

[4] S. Han, H. Mao and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[5] M. Courbariaux, Y. Bengio and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," *Advances in neural information processing systems,* p. 3123–3131, 2015.

[6] W. Chen, J. Wilson, S. Tyree, K. Weinberger and Y. Chen, "Compressing neural networks with the hashing trick," *International Conference on Machine Learning,* p. 2285–2294, 2015.

[7] R. Ding, Z. Liu, R. Shi, D. Marculescu, R. D. Blanton, L. Ding, Z. Liu, R. Shi, D. Marculescu and R. D. Blanton, "LightNN: Filling the Gap between Conventional Deep Neural Networks and Binarized Networks," *Proceedings of the on Great Lakes Symposium on VLSI 2017, ACM,* p. 35–40, 2017.

[8] V. Sindhwani, T. Sainath and S. Kumar, "Structured transforms for small-footprint deep learning," *Advances in Neural Information Processing Systems,* p. 3088–3096, 2015.

[9] J. Lee, M. Stanley, A. Spanias and C. Tepedelenlioglu, "Integrating machine learning in embedded sensor systems for Internet-of-Things applications," *IEEE International Symposium on Signal Processing and Information Technology (ISSPIT), Limassol,* pp. 290-294, 2016.

[10] L. Lai, N. Suda and V. Chandra, "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs," in *arXiv.org*, 2018.

[11] B. Scholkopf, R. Williamson, A. Smola, J. Shawe-Taylort and J. Platt, "Support vector method for novelty detection," *NIPS'99 Proceedings of the 12th International Conference on Neural Information Processing Systems,* pp. Pages 582-588, 4 December 1999.

[12] J. Goulermas, A. Findlow, C. Nester, P. Liatsis, X.-J. Zeng, L. Kenney, P. Tresadern, S. Thies and D. Howard, "An instance-based algorithm with auxiliary similarity in- formation for the estimation of gait kinematics from wearable sensors," *IEEE Trans. Neural Netw.,* vol. 19, p. 1574–1582, 2008.

[13] J. Zhang, T. Lockhart and R. Soangra, "Classifying lower extremity muscle fatigue during walking using machine learning and inertial sensors," *Annu. Biomed. Eng.,* vol. 42, pp. 600-612, 2015.

[14] J. Paulo, P. Peixoto and U. J. Nunes, "ISR-AIWALKER: Robotic Walker for Intuitive and Safe Mobility Assistance and Gait Analysis," *IEEE Trans. Human-Machine Syst.,* vol. 47 no. 6, p. 1110–1122, 2017.

[15] M. Yang, H. Zheng, H. Wang, S. McClean, J. Hall and N. Harris, "A machine learning approach to assessing gait patterns for complex regional pain syndrome," *Med. Eng. Phys.,* vol. 34 (6), pp. 740-746, 2012.

[16] Rajapakse, A. R. Omondi and J. C., "FPGA implementation of neural network," 2006.

[17] H. G. E. and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science,* pp. Vol. 313. no. 5786, pp. 504 - 507, 28 July 2006.

[18] Urbanowicz, R. J., Moore and J. H., "Learning Classifier Systems: A Complete Introduction, Review, and Roadmap," *Journal of Artificial Evolution and Applications,* pp. 1-25, 2009 Sept. 22.

[19] C. Zhang and S. Zhang, Association rule mining: models and algorithms, Springer-Verlag, 2002.

[20] D. Castro, L. Nunes and J. Timmis, Artificial immune systems: a new computational intelligence approach, Springer Science & Business Media, 2002.

[21] Z. Cui, W. Chen and Y. Chen, "Multi-Scale Convolutional Neural Networks for Time Series Classification," in *Computer Vision and Pattern Recognition*, 2016.

[22] Y. Roh, G. Heo and S. E. Whang, "A Survey on Data Collection for Machine Learning: a Big Data - AI Integration Perspective," IEEE, 2018.

[23] R. J. H. a. G. Athanasopoulos, "Forecasting: Principles and Practice," *OTexts,* 2012.

[24] N. S. Madiraju, S. M. Sadat, D. Fisher and H. Karimabadi, "Deep Temporal Clustering : Fully Unsupervised Learning of Time-Domain Features," 2018.

[25] H. Jaeger, "Echo state network," 2007. [Online]. Available: http://www.scholarpedia.org/article/Echo_state_network.

[26] M. Lukosevicius, H. Jaeger and B. Schrauwen, "Reservoir computing trends," *KI,* vol. 26(4), p. 365–371, 2012.

[27] E. T. S.-B. P. Holmes, "Stability".*Scholarpedia.*

[28] J. D. X. D. a. B. S. David Verstraeten, "Memory versus non-linearity in reservoirs," in *International Joint Conference on Neural Networks, IJCNN 2010*, Barcelona Spain, 2010.

[29] J. Schmidhuber, "A fixed size storage O(n3) time complexity learning algorithm for fully recurrent continually running networks," *Neural Computation,* vol. 4, no. 2, pp. 243-248, 1992.

[30] P. Y. S. a. P. F. Yoshua Bengio, "Learning longterm dependencies with gradient descent is difficult.," *IEEE Trans. Neural Networks,* pp. 157-166, 1994.

[31] M. C. a. P. Tiño, "Comparison of echo state networks with simple recurrent networks and variable-length markov models on symbolic sequences," in *Artificial Neural Networks - ICANN*, Porto, Portugal, 2007.

[32] G. Ditzler and al., "Learning in nonstationary environments: A survey," *IEEE Computational Intelligence Magazine 10.4,* pp. 12-25, 2015.

[33] C. Alippi, G. Boracchi and M. Roveri, "A just-in-time adaptive classification system based on the intersection of confidence intervals rule," *Neural Networks 24.8,* pp. 791-800, 2011.

[34] D. Picard, "Testing and estimating change-points in time series," *Advances in applied probability 17.4,* pp. 841-867, 1985.

[35] P. Q. a. C. W. K. D. M. Hawkins, "The changepoint model for statistical process control," *Journal of Quality Technology,* vol. 35, no. 4, pp. 355-366, 2003.

[36] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics,* pp. 50-60, 1947.

[37] Q. a. L. Y. a. C. Y. a. K. Y. a. C. T. a. Y. H. Yang, "Federated learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning,* vol. 13, no. 3, pp. 1-207, 2019.