SEMIoTICS

Deliverable D4.3
Embedded Intelligence and Local Analytics (first draft)

Deliverable release date	31/05/2019, revised (25/11/2019)
Authors	1. Danilo Pau, Mirko Falchetto (ST), 2. Darko Anicic, Arne Broering (SAG), 3. Domenico Presenza (ENG) 4. David Parra (UP) 5. Lukasz Ciechomski (BS) 6. Christos Tzagkarakis (FORTH)
Responsible person	Danilo Pau, Mirko Falchetto (ST)
Reviewed by	Konstantinos Fysarakis (STS), Nikolaos Petroulakis (FORTH)
Approved by	PTC Members (Vivek Kulkarni, Nikolaos Petroulakis, Ermin Sakic, Mirko Falchetto, Domenico Presenza, Verikoukis Christos) PCC Members (Vivek Kulkarni, Nikolaos Petroulakis, Verikoukis Christos, Georgios Spanoudakis, Domenico Presenza, Danilo Pau, Joachim Posegga, Darek Dober, Kostas Ramantas, Ulrich Hansen)
Status of the Document	Final (Revised)
Version	1.0 - Amended
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
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
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
KWS	KeyWord Spotting
LA	Local Analytics
LIDAR	LIght Detection And Ranging
LSTM	Long Short Term Memory
MCU	Micro Controller Unit

MHz	Mega Hertz (unit of frequency)
ML	Machine Learning
NPU	Neural Network Processing Unit
RBML	Rule Based Machine Learning
RGB-D	Red Green Blue Depth (images)
RNN	Recurrent Neural Network
SARA	Socially Assistive Robotic Solution for Ambient assisted living
SAS	Self-Adaptation System
SoC	System on Chip
SOTA	State Of The Art
SVM	Support Vector Machine
SW	SoftWare
UCx	Use Case x

1. INTRODUCTION

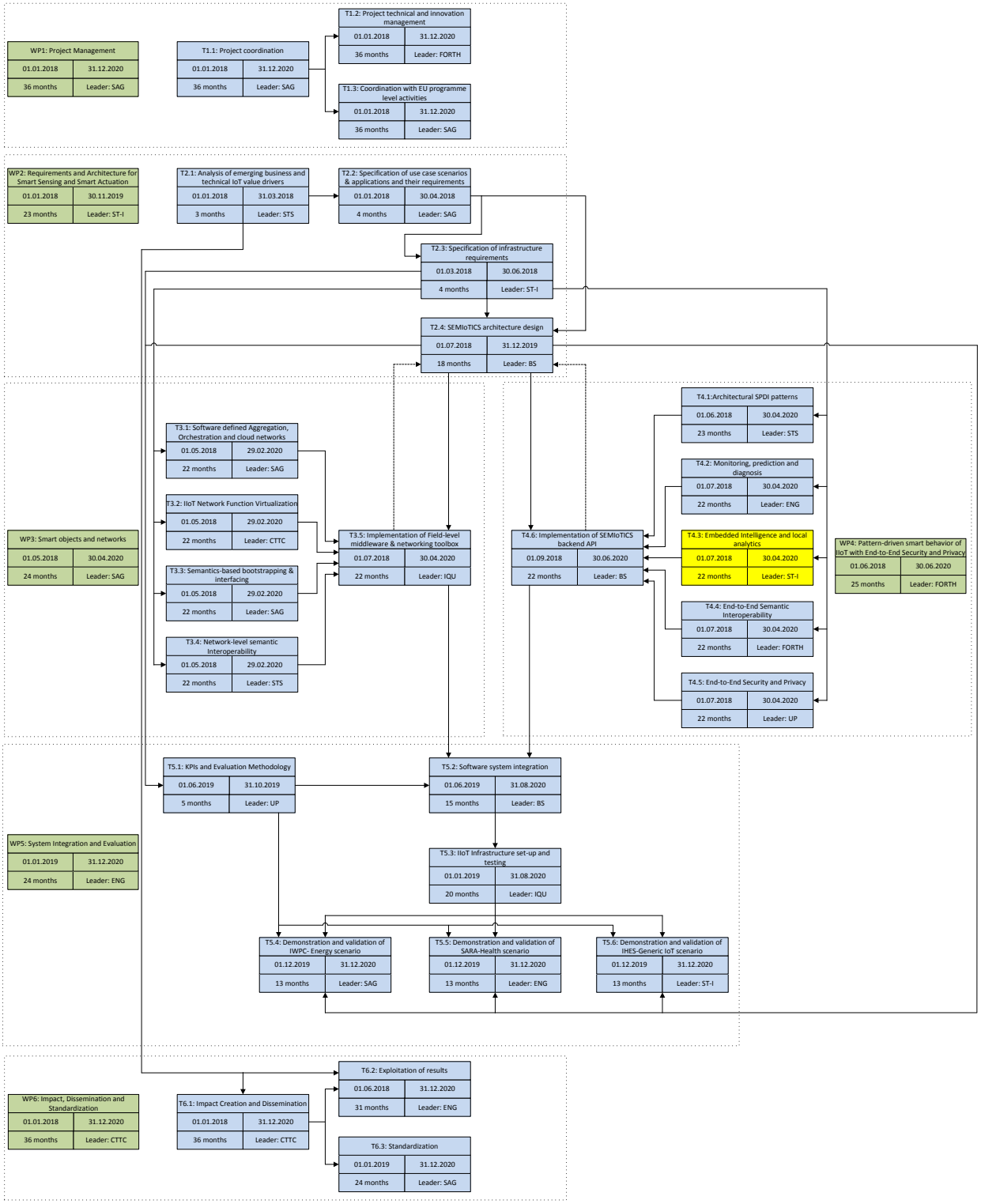
This deliverable is the first output of Task 4.3 – "Embedded Intelligence and local analytics" and, as such, it focuses on the building blocks required for introducing embedded intelligence at the IIoT/IoT Field Device Level. More specifically, it aims to provide an initial definition of Local Analytics (LA) embedded mechanisms, so as to enable semi-autonomic local reaction/adaptation within IIoT/IoT devices. This deliverable exploit part of the work done on D4.2, where a preliminary analysis of the concepts related to analytics has been identified and defined. In D4.2 this initial research has been exploited to identify promising analytics algorithms deployable at all levels of the architecture, with the specific goal of identifying the most relevant ones to be used for defining the monitoring component of the architecture. On the other hand, the initial research done in this deliverable D4.3 identifies a subset of those algorithms, addressing time series processing from sensing data, suitable for a real *lightweight* deployment at the Field Device level on very constrained devices. The main outcomes of this deliverable will be used to identify the initial set of algorithms that will be fully characterized in D4.10 as part of the Local Embedded Analytics Component.

The content of this deliverable is organized in order to provide a clear overview and promising approaches existing todays in literature or as common practices for local analytics algorithms. An overview of them are thus provided on section 2 and 3 of the deliverables with the intent to have a complete overview of today's technical landscape in order to identify the subset of algorithms and tools that could be mapped into SEMIoTICS framework. Thus, not all of them will be integrated as part of the SEMIoTICS components but only the ones that are presented in subsections 2.3 and 3.4.

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

- Section 2 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 to define 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 specifies in more detail the intended scenario together with a preliminary introduction of a typical ML algorithm development and deployment workflow using Deep Learning tools, from a perspective of the specific challenges about the deployment at Field Device level. Regarding the DL tools, an exhaustive overview of the same is provided in this deliverable: only a subset of them will be adopted and sponsored in SEMIoTICS as key components that facilitates the integration of proper AI/ML solutions into the reference framework. Also depending on the specific use case scenario involved, a tool is more suitable for adoption than another one: the final selection will be provided in D4.10 after proper development and deployment will be consolidated.
- Finally, Section 4 derives the conclusions of the deliverable, the current implementation status and the next planned steps.

1.1. PERT chart of SEMIoTICS



Please note that the PERT chart is kept on task level for better readability.

2. EMBEDDED INTELLIGENCE IN IOT/IIOT ENVIRONMENTS

2.1. The SEMIoTICS Embedded Intelligence vision

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 D2.4 – SEMIoTICS Architecture for further details) where embedded intelligence algorithms will be deployed. These components will export coherent APIs, specific at each logical level of the architecture, in order 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 1 the generic approach envisaged within the SEMIoTICS framework is visualised. 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 done also at IoT/IIoT Gateway in SEMIoTICS, mainly for supporting monitoring of the SEMIoTICS Field Devices, but also part of the analytics that could not be deployed at lowers level of the architecture. Please refer to D4.2 for additional details about these implementation details at IoT/IIoT Gateway level.

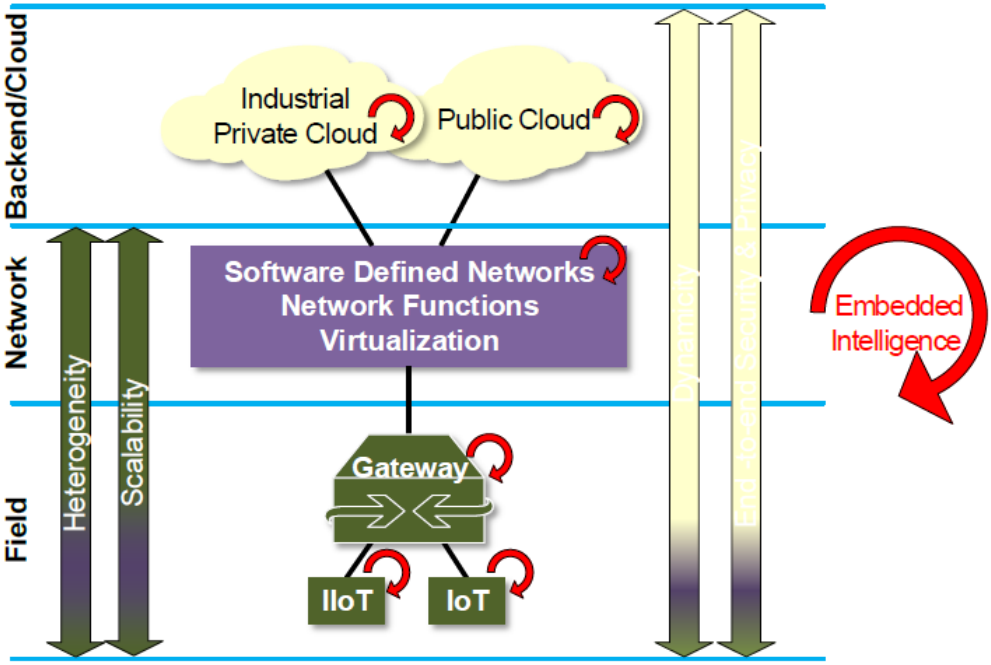


FIGURE 1: MULTI-LAYERED INTELLIGENCE IN SEMIOTICS

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 this kind 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 the 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 2). The fundamental concern of the Internet of Things (IoTs) 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 battery 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

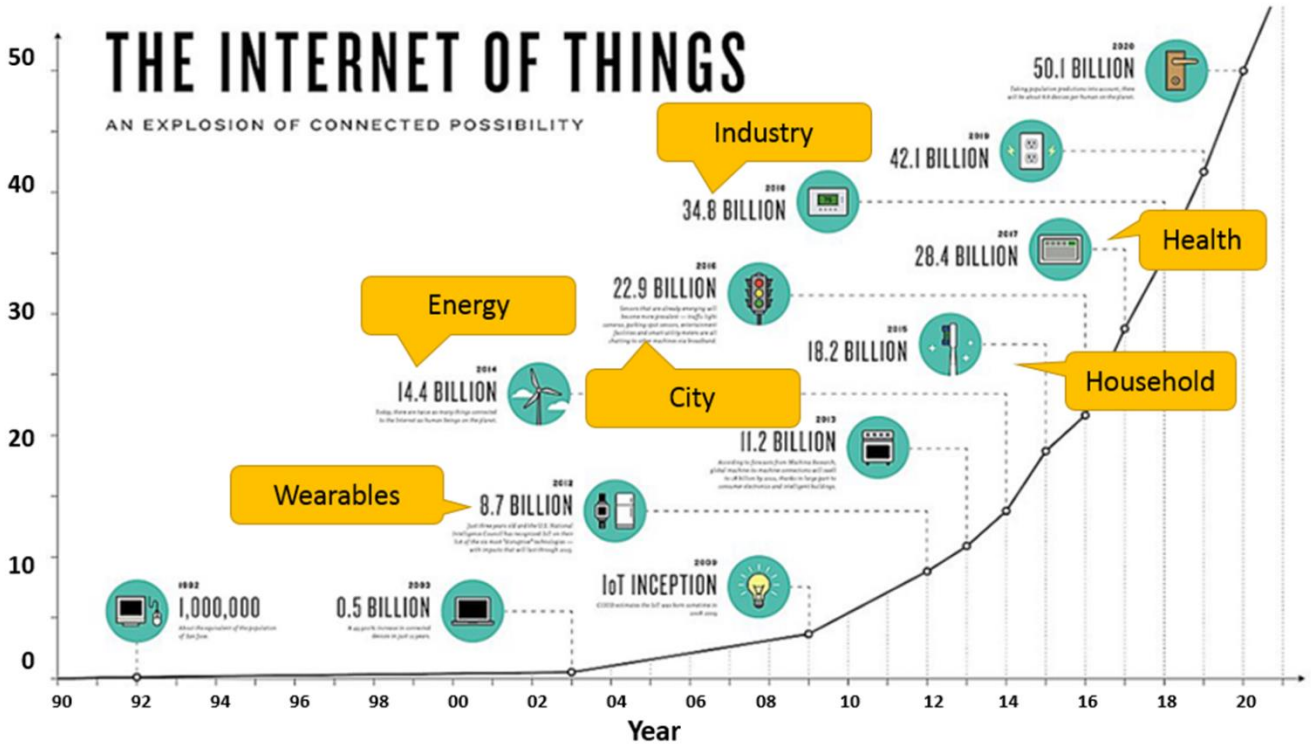


FIGURE 2: FIGURE 2 IIOT/IOT CONNECTED DEVICES GROWTH[1]

increasing is if they have batteries that last a very long time, or if they can use energy harvesting (like solar cells from indoor lighting).

Constraining energy aims to keep energy usage at the milliwatt 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 milliwatt 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.
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 milliwatt, even down to tens of microwatts. The same is true for general Microprocessors and Digital Signal Processors (DSPs) that can

[1] http://zolertia.sourceforge.net/wiki/images/e/e8/Z1_RevC_Datasheet.pdf

process tens or hundreds of millions of calculations for under a milliwatt, 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 analysed at all. This is the point where SEMIoTICS, developing the concept of multi-layered intelligence, aims at bringing a novel approach and skills to analyse 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 3 below. This figure shows a typical complete 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 analyse 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*.

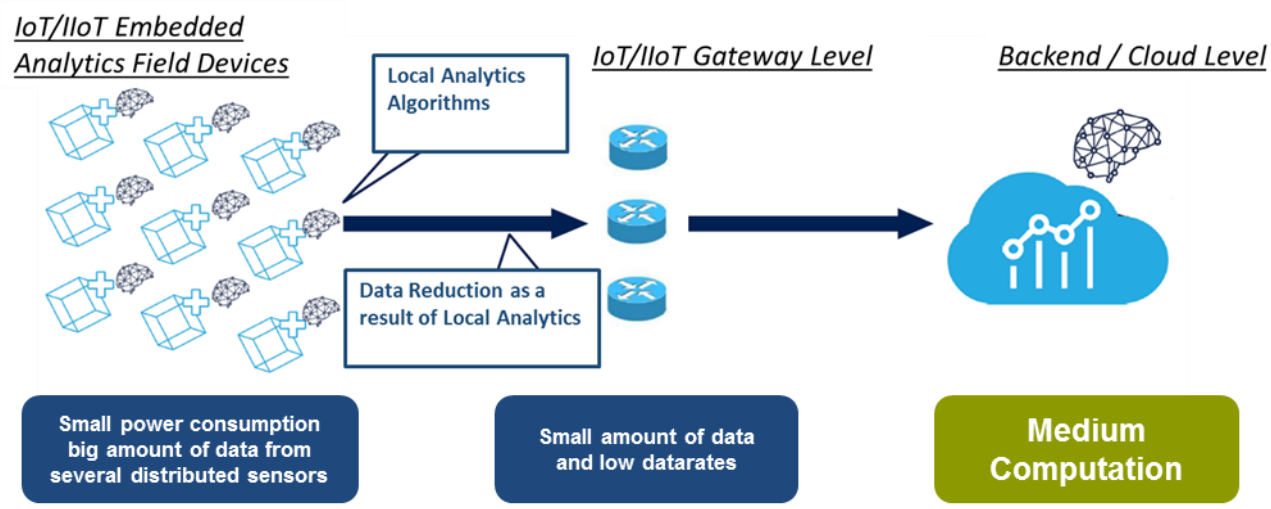


FIGURE 3: SEMIOTICS LOCAL ANALYTICS FLOW

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, D4.3 focuses on local embedded intelligence as mentioned above. 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.

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:

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

- 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 W samples acquired at times [N-1...N-W-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 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 need 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 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 behaviour: 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 behaviour 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 analyse 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 behaviour 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.

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.

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

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

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 SEMIoTICS framework, forming 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 Neighbours algorithm for devices with limited storage.
The described approach is inspired by k-Nearest Neighbours 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 CubeMX middleware software.

2.3.3. IIOT/IOT FIELD DEVICES UNSUPERVISED LOCAL ANALYTICS

A major contribution in consolidating these local embedded analytics approaches, addressing real life problems, will be provided by all three major use cases identified in D2.2 within SEMIoTICS and the requirements described in the D2.3, but a strong emphasis on this specific aspect will be given by the IHES demonstrator. This embedded engineered system will be deployed at the field device level by defining a set of specific lightweight algorithms deployable on heavily constrained STM32 MCUs devices (D2.3-Table8-R.UC3.3). These algorithms will be wrapped as a dedicated “Local Embedded Analytics” component, in order to provide a validated example of an appliance in the IIoT/IoT domain. This system will provide advanced enabling technologies and system libraries to implement semi-automatic unsupervised local adaption, borrowing some concept from these research domains. A self-adaptive system (SAS) is a system able to adapt its behaviour 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.4).

All the algorithms used in the proposed Local Embedded Analytics and for the semi-automatic unsupervised local adaption of STM32 device are based on the requirements specified in D2.3. This section shortly summarizes these requirements. The STM32 MCUs devices should embed sensors, obtain data and process this raw data using statistical methods and ML algorithms. This approach aims to minimize the traffic of raw data if favour of a communication triggered by events. This must be based on secure 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 devices (i.e., a set of smart MCUs equipped with communication capabilities), taking advantage of the ability to autonomously detect changes in

[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

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 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.UC3.8).
2. Apply the learnt model to compute a residual signal using input samples and predictions (D2.3-Table8-R.UC3.4).
3. Design a model-free change detection test applied to residual signal (D2.3-Table8-R.UC3.5).
4. Design a change-point method to validate the detected change (D2.3-Table8-R.UC3.5).

A technological implementation of the proposed methodology encompassing linear predictive models, autoregressive models, specifically designed AI/ML nonlinear predictive algorithms (see section 3.4.1 for further details) and some specific tailored change detection algorithms with a chained validation phase to mitigate false positive changes are currently under design and initial mock-up functional implementation. The final definition, specific deployment, testing and the precise set of algorithms adopted in a single MCU unit will be detailed later in D4.10, which is the final draft of this deliverable. This generic data streams processing pipeline will be completely mapped on ST MCU STM32 devices as a dedicated C library exposing dedicated APIs as part of the activities related to the UC3 demo scenario.

This mapping demonstrates how it is possible to implement an efficient yet generic data reduction 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 4.

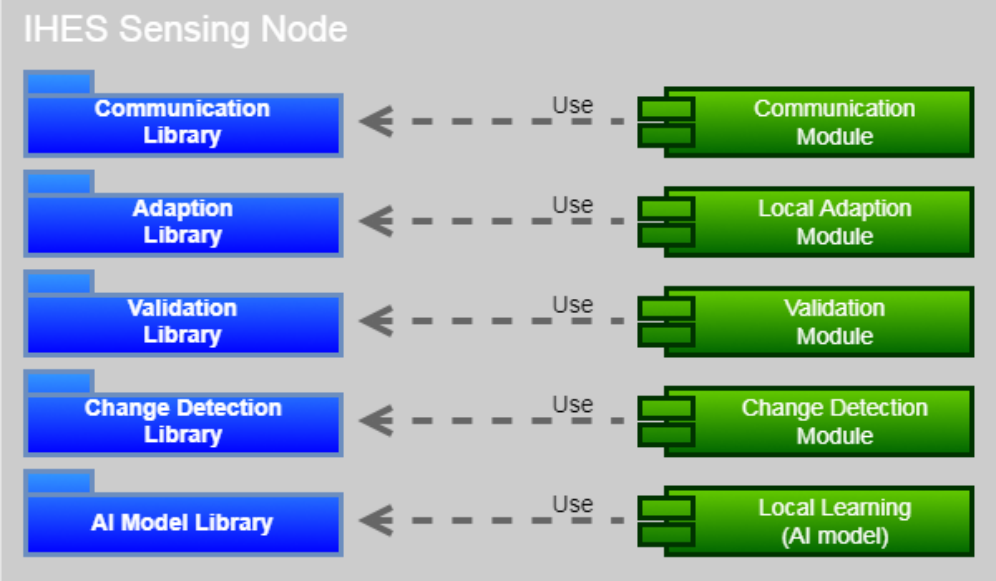


FIGURE 4: LOCAL LEARNING / ADAPTATION ALGORITHMS - IHES NODE

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.

The novelty of the proposed methodology applied to low-cost resource constrained micro controllers, both in terms of computational power and integrated RAM and ROM memories, resides in the joint use of:

- A learning mechanism to build the predictive model describing the sensor data stream over time.
- A 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 estimates by themselves 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.

The whole processing pipeline could be represented as a state machine composed by initialization / learning and runtime modes operating over a shared global context. 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. 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 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). A similar approach for the change detection can be seen in the paper [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 *"to never to solve a problem which is more general than the one we actually need to solve as an intermediate step"*.

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 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 neighbours (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 didn’t have large amounts of data to train them. Examples of the ANN approach related to the ML can be found in Figure 5.

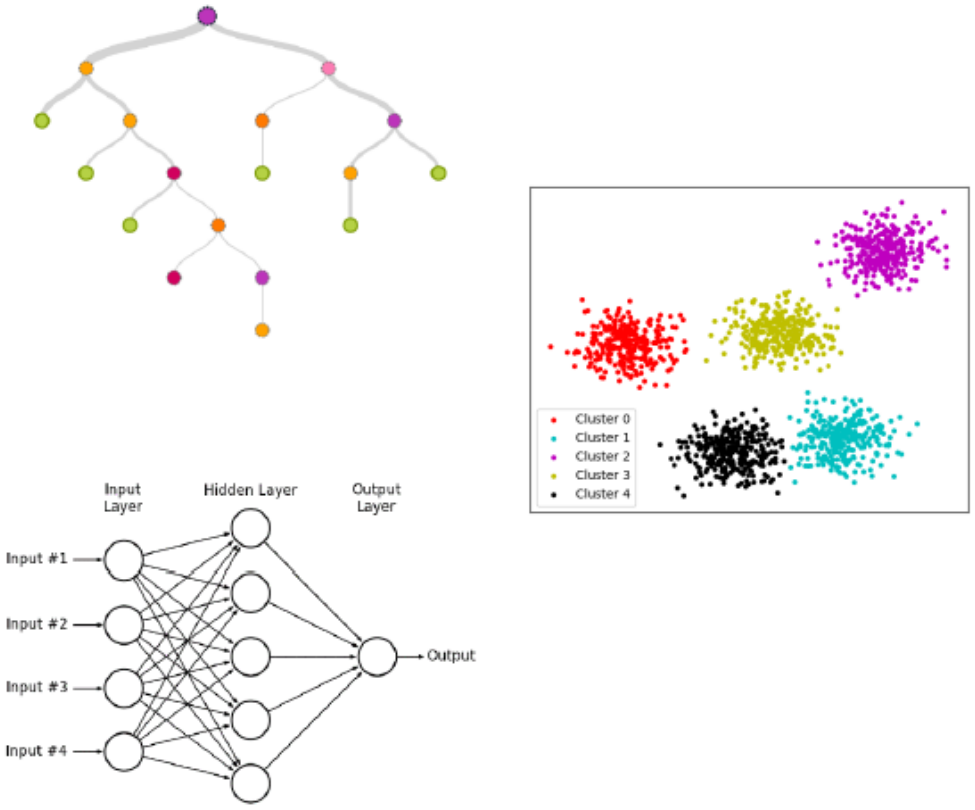


FIGURE 5: ANN APPROACH RELATED TO THE ML

More precisely, looking at Figure 6. 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 train on past examples to build and estimate models. ML usually is 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

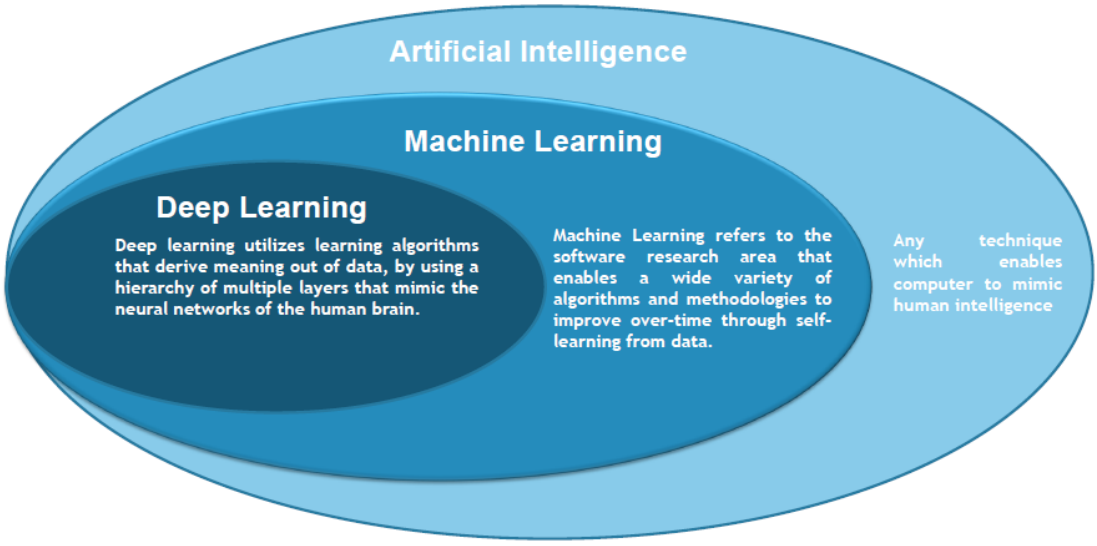


FIGURE 6: ARTIFICIAL INTELLIGENCE DOMAINS CLASSIFICATION

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, unlabelled 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 analysing 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 sceptical 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 few years ago, 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, 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 approaches to training - 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. 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 is more indicated for 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 has 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 network. In the last couple of years, they

[10] Theano DL Tool - http://www.deeplearning.net/software/theano/
[11] Caffe DL Tool - https://caffe.berkeleyvision.org/

have been flanked and surpassed in term of features and usability by other 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 behaviour 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. This trend is already confirmed by Intel, 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 7.

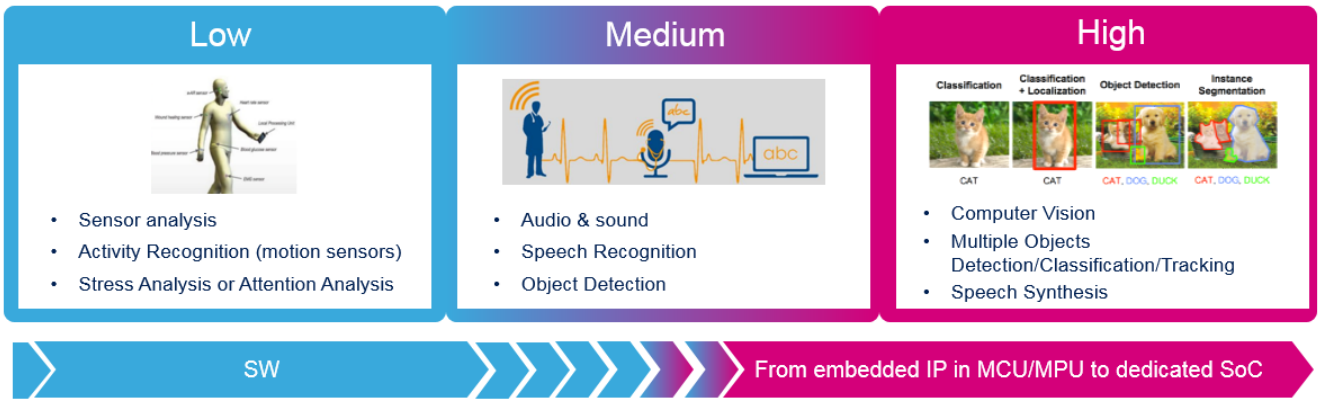


FIGURE 7: FIGURE 7 AI SOLUTIONS VS APPLICATION REQUIREMENTS[18]

In SEMIoTICS, we are interested in investigating the feasibility of deploying the low end of appliances depicted in Figure 7 as part of the Embedded Local Analytics Component. There are several reasons that motivates it. The main one is that we are focusing on the deployment on 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

[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

physical actuators or process data from simple sensors. So, 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 algorithm 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 which 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[20] 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 analyse 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.

- **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 on 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 datacentres 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[21]), 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.

[20] https://en.wikipedia.org/wiki/Comparison_of_deep-learning_software
[21] https://www.tensorflow.org/lite

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 8.

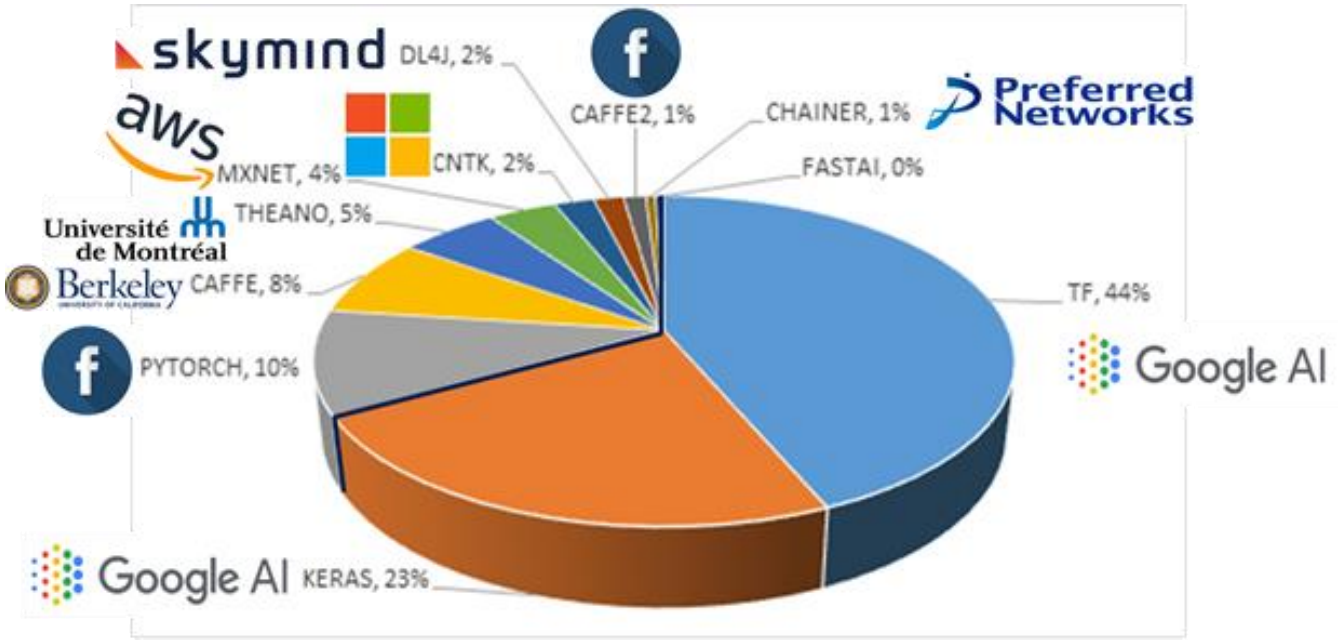


FIGURE 8: FIGURE 8 DEEP LEARNING TOOLS POPULARITY[22]

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. For supervised learning they need to be additionally annotated, whereas with unsupervised algorithms they need not to be. 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 cases (e.g. in audio / inertial processing) some pre-processing algorithm should be designed in order to transform the input data in a more convenient data representation. 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

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

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 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[24], 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 9, 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 distributed local embedded 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 updated to support the selected ML algorithm.

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

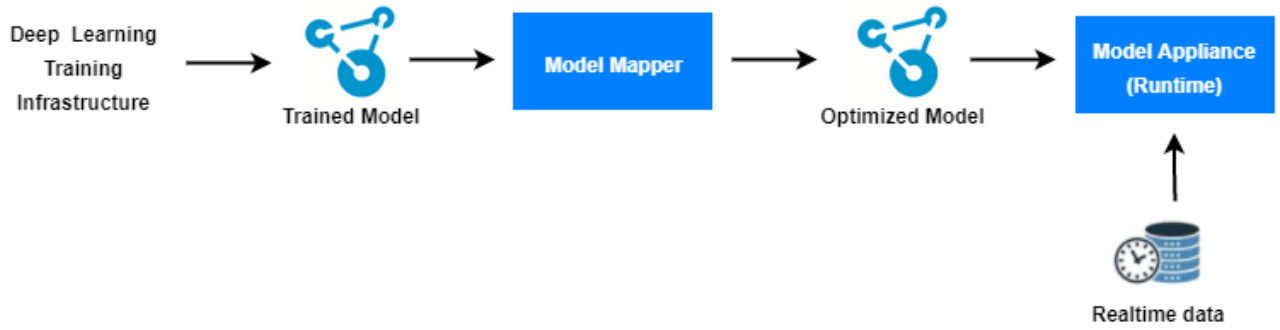


FIGURE 9: DEEP LEARNING TOOLS DEPLOYMENT FLOW

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 DCNN, RNN, and 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 in D4.2 for more algorithmic details). In terms of neural networks, we applied an auto encoder as developed in this Matlab auto encoder example[25].

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

The STM32Cube.AI[26] 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 Convolutional Neural Networks (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 2). 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 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

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

neural network model, developed by the user exploiting one of the most popular off-the-shelf neural networks DL design tools, and then it automatically generates an optimized neural network C code library to run on the STM32'powered platform of choice as bare metal FW, giving the ability to test the defined deep learning solution on the field in a very fast and efficient fashion.

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

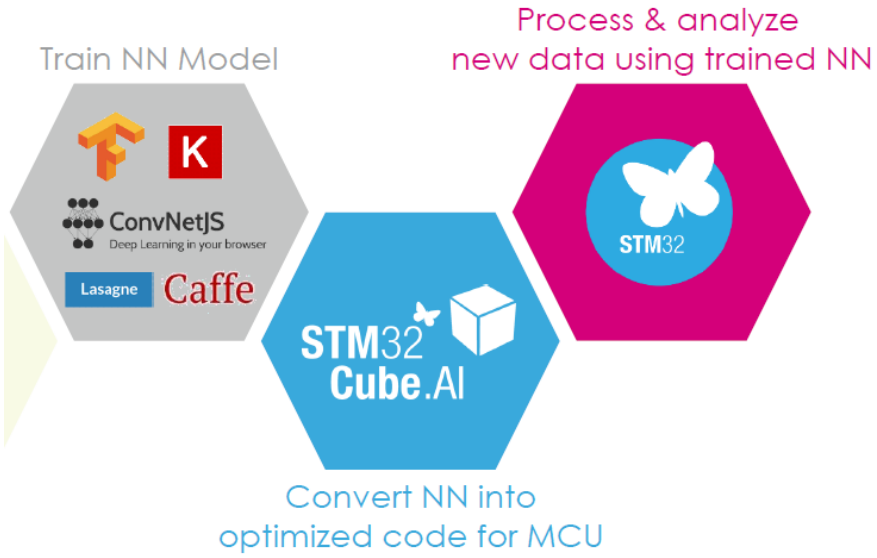


FIGURE 10: FIGURE 10 CUBE.AI TOOL PLUGIN

X-Nucleo Boards[27], BlueCoin[28] or SensorTile[29] 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.

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

3.4. IIoT/IoT Field Devices AI Algorithms

In the previous subsections, a collective description of some algorithmic approaches, from SOTA analysis, has been presented: they were about regression models and embedded intelligence, and the support tools that could facilitate their design. In the following sub-sections, a presentation of some more specific algorithms, centred on the definition of the SEMIoTICS UCs, as declared in D2.2, is provided. These algorithms will be the basis forming the initial version of the Local Embedded Analytics Component. As a result of the next deployment and implementation activities (planned during the cycle 2 and cycle 3 interactions), this D4.3 SOTA analysis on algorithms and DL tools will allow to extend/combine new algorithmic ideas that may be needed to complete the UC definition. The result of these activities with a detailed reporting of the final algorithmic set adopted will be reported in the final draft of this deliverable (i.e. D4.10). This methodology is 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 objectives claimed by the project.

An important subset of AI/ML algorithms, implemented as part of the Local Embedded Analytics component at the Field Device level, will be used to support the SEMIoTICS scenarios 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.

3.4.1. TIME SERIES PREDICTION

In the IHES system-horizontal demonstrator (UC3), a 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[30] (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 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 ESN models, first introduced by Jaeger [24], are a type of AI/ ML network architecture based on Reservoir Computing[31] (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 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. 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

[30] https://en.wikipedia.org/wiki/Recurrent_neural_network
[31] https://en.wikipedia.org/wiki/Reservoir_computing

it is not involved in the online training phase. The ESN model selected as initial reference for the implementation of the time series prediction could be formally defined and visually represented as follows:

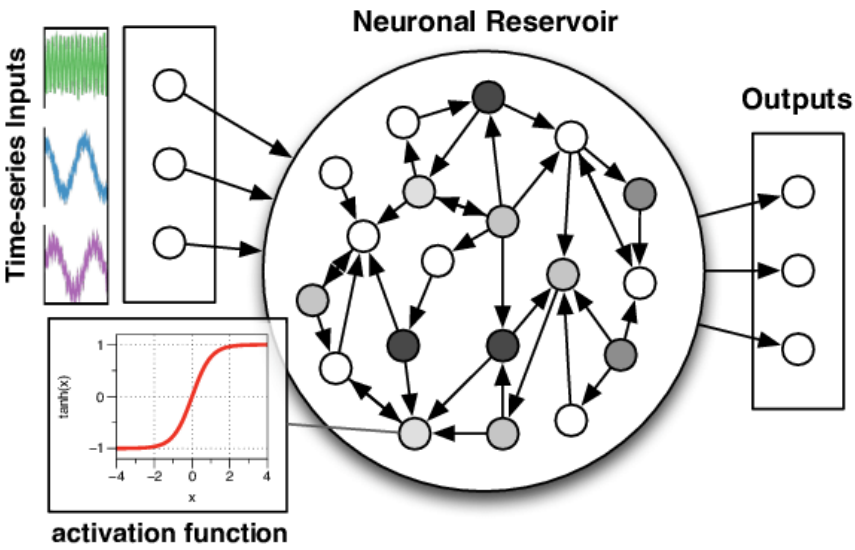


FIGURE 11: ESN MODEL FOR TIME SERIES[32]

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 [25].
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 neuronal reservoir and input weight matrices
- $\boldsymbol{b}$ represents the bias factor introduced
- $f()$ is the non-linear activation function of neurons
- The input and the hidden state at the time instant $\boldsymbol{n}$ 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.

3.4.2. 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 has the 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 at work, where the algorithm runs at the node level processing sensed data.

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

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 12). The two-time series (i.e. left leg distance and right leg distance) are the inputs for the gait analysis algorithm.

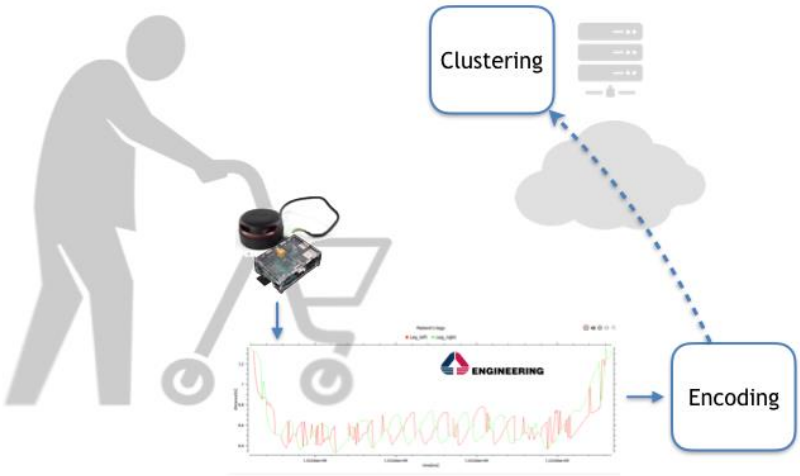


FIGURE 12: GAIT ANALYSIS SETTING IN SARA UC

As a first step, 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 [26] 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 13 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.

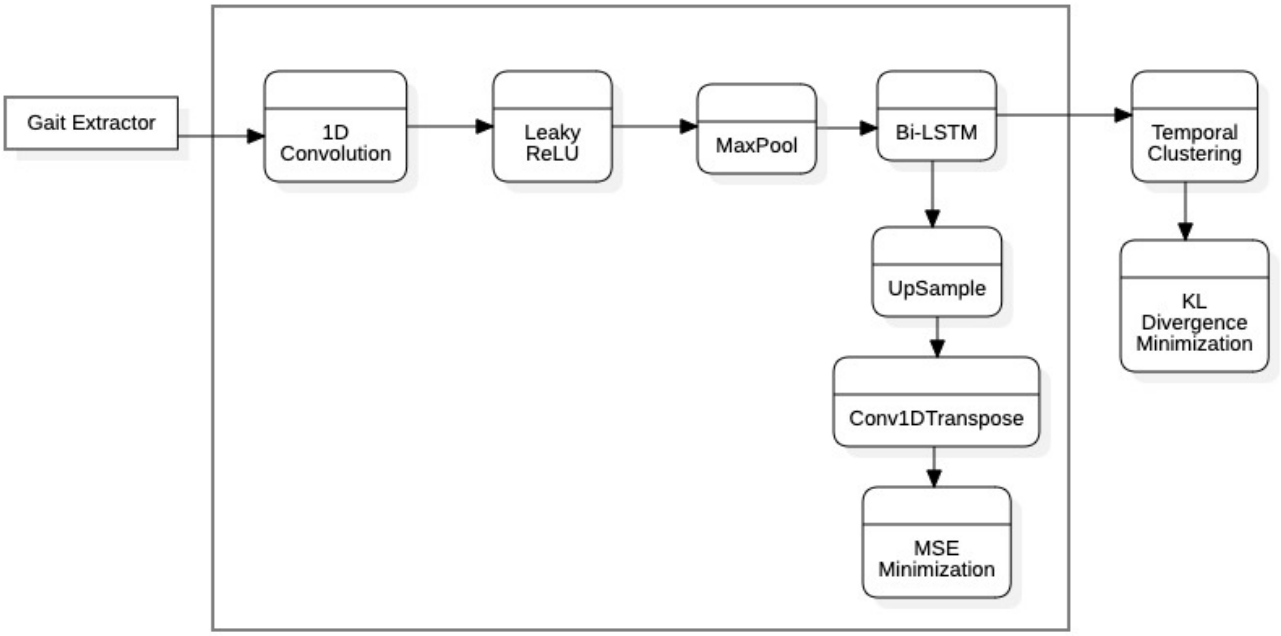


FIGURE 13: DEEP TEMPORAL CLUSTERING (DTC) DATAFLOW

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 encoding is 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 form 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 [26].

4. CONCLUSION AND FUTURE WORK

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 backend for further processing, if needed. In a nutshell, our approach ensures that devices can save energy overall, protect the end user's privacy, and support massive scalability of the infrastructure by spreading these local analytics 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 of some ML tools and possible ways to design and implement more lightweight versions of some algorithms for constrained devices. The range of possibilities for statistical and ML algorithms modelling are 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 3.4. Moreover, the base for the local analytics functionality is focused in two aspects: time series prediction, and time series clustering. These approaches provide support for local embedded intelligence and local analytics within the project for the processing of time varying signals. They are provided as working functional modules that have been identified as part of the cycle1 activities. For some of them, a functional mock-up of their main functionalities has been already mapped on a selected target platform of interest. During cycle 2 activities, we plan to complete the implementation of all required functions and functionalities and to integrate them in the whole framework during the integration phase, in cycle 3. Between the cycle 2 and cycle 3 interactions, the project will further characterize the algorithms to ensure that all the requirements are fulfilled. Consequently, fine-tuning and testing of the components are planned between the cycle 2 and cycle 3 integration activities, where some other algorithm could eventually be added depending on emerging needs from the use cases. These algorithms are likely to be evaluated from the set declared in D4.2 *"SEMIoTICS Monitoring, prediction and diagnosis mechanisms"* where all the algorithms used in SEMIoTICS are described.

As this document is a preliminary draft, we will provide a full report and characterization on the final implementation of the mechanisms highlighted in this document, in particular regarding algorithms presented in sections 2.3.1, 2.3.2, 2.3.3, 3.4.1 and 3.4.2. These aspects are covered by the final draft of this document (i.e. D4.10), due by M26 in the project.

5. 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] H. Jaeger and H. Haas, “Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication,” *Science,* vol. 304, no. 5667, pp. 78-80, 2004.

[24] H. Jaeger, “Echo state network,” 2007. [Online]. Available: http://www.scholarpedia.org/article/Echo_state_network.
[25] M. Lukosevicius, H. Jaeger and B. Schrauwen, “Reservoir computing trends,” *KI,* vol. 26(4), p. 365–371, 2012.
[26] N. S. Madiraju, S. M. Sadat, D. Fisher and H. Karimabadi, “Deep Temporal Clustering : Fully Unsupervised Learning of Time-Domain Features,” 2018.
[27] B. Jin, Y. Chen, D. Li, K. Poolla and A. L. Sangiovanni-Vincentelli, “A One-Class Support Vector Machine Calibration Method for Time Series Change Point Detection,” in *arxiv 1902.06361*, 2019.