BRAIN-IoT

model-Based fRamework for dependable sensing and Actuation in INtelligent decentralized IoT systems

D3.2 – Initial AI and ML features for smart behaviour and actuation

Deliverable ID	D3.2
Deliverable Title	Initial AI and ML features for smart behaviour and actuation
Work Package	WP3
Dissemination Level	PUBLIC
Version	1.0
Date	2019-02-28
Status	Final
Lead Editor	IM
Main Contributors	Víctor Sóñora (IM)

Published by the BRAIN-IoT Consortium

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 780089.

Document History

Version	Date	Author(s)	Description
0.1	2018-10-19	Víctor Sóñora (IM)	First Draft with TOC.
0.2	2018-11-15	Diego Fernández (EMALCSA)	Draft with use cases reviewed.
0.3	2018-11-23	Víctor Sóñora (IM)	New content for both Introduction and Smart Behaviours definition. Review the EMALCSA use cases. Content for the ongoing analysis with Paremus regarding Smart Behaviours packaging.
0.4	2019-01-09	Víctor Sóñora (IM)	Rewrite Runtime Environment section, according to recent agreements.
0.5	2019-01-14	Miguel Boubeta (IM)	Full review, corrections and suggestions.
0.6	2019-01-15	Víctor Sóñora (IM)	Full review.
0.7	2019-01-24	Víctor Sóñora (IM)	Updates to reflect feedback provided by Paremus.
0.8	2019-01-29	Víctor Sóñora (IM)	Updates and fixes thanks to feedback provided by Tim Verbelen (external review via Paremus).
1.0	2019-02-13	Víctor Sóñora (IM)	Updates and fixes following feedback from STM.

Review History

Version	Review Date	Reviewer	Summary of Comments
1.0	2019-01-28	Richard Nicholson, Tim Ward (Paremus)	Approved with minor comments.
1.0	2019-02-13	Mario Diaz Nava (STM-GNB)	Approved with minor comments.

Table of Contents

Document History ... 2
Review History ... 2
Table of Contents ... 3
1 Introduction ... 4
1.1 Scope and goals ... 4
1.2 Related documents ... 4
2 Smart Behaviour definition and study ... 5
2.1 Context, definitions: AI and ML ... 5
2.2 Classic AI and smart agent system approach ... 7
2.3 ML methods: a basic useful taxonomy ... 9
2.4 Considerations about Artificial Neural Networks and Deep Learning state of the art ... 10
2.5 Study of well-designed ML libraries ... 12
3 Brain-IoT Use Cases analysis for Smart Behaviours ... 13
3.1 Water Critical Infrastructure use cases and scenarios ... 13
3.2 Service Robotics use cases and scenarios ... 15
4 Solution Manager System ... 17
4.1 Analysis to extract AI and ML behaviours from current BRAIN-IoT Use Cases ... 17
4.2 Proposed design: overall view ... 19
4.3 Decoupling inputs/outputs interface from behaviour implementation ... 20
4.4 Training and production environments, support for different lifecycles ... 21
4.5 Machine Learning Solution Subsystem unique elements ... 22
4.6 Different treatment and requirements regarding datasets ... 22
4.7 Assumed limitations, adopted conventions, pending work ... 22
5 Design, modelling and formal description of Smart Behaviours ... 24
5.1 Smart Behaviour elements and their visibility ... 24
5.2 Some considerations about modelling, formal verification and Smart Behaviours ... 25
5.3 ML Training to production workflow: a small topology ... 25
6 Runtime Environment and integration with Brain-IoT architecture ... 27
6.1 BRAIN-IoT architecture introduction and considerations ... 27
6.2 AI / ML elements runtime implementation ... 28
6.3 AI / ML data exchange formats ... 29
6.4 Smart Behaviour packaging: options and proposed solution ... 30
7 Conclusions ... 34
Appendix ... 35
A brief look at successful ML technologies ... 35
Acronyms ... 37
List of figures ... 37
List of tables ... 38
References ... 38

1 Introduction

This document is a deliverable of the BRAIN-IoT project, funded by the European Commission, under Horizon 2020 Research and Innovation Program (H2020). It belongs to WP3 – IoT Framework for smart dynamic behaviour, under Task 3.2 – AI and ML features for smart behaviour and actuation.

1.1 Scope and goals

The main role of this task is to design and implement the features that deal with Artificial Intelligence (AI) and Machine Learning (ML) techniques in Smart Behaviours as they are being defined in BRAIN-IoT.

The two main sets of use-case real world scenarios, service robotics and critical infrastructure monitoring, will dictate the specific features where some applied intelligence (analysis, prediction, collaborative context base behaviour) is needed to solve the problems.

However, it is a main goal of this document to generalize the solutions available for this kind of intelligence. A research in classic and state-of-the-art AI and ML methods will be elaborated to in order to design for a set of abstractions that can be used under a generalized approach. In such a way that this proposed design:

- Solves the use-case scenarios.
- Covers a wide enough range of problems in the context of Smart Behaviours in distributed environments.
- Is well defined in terms of the ongoing IoT-ML definition elements.
- Explore the Capabilities that would be needed to advertise regarding Smart Behaviours requirements.
- Outline which Smart Behaviours would benefit from the exogenous coordination approach advocated and developed in BIP.
- Those elements can be modelled and managed within the Brain-IoT developed modelling tools.

1.2 Related documents

ID	Title	Reference	Version	Date
D3.1	Initial data and capabilities models for cross-platform interoperability		1.1	2018-10-17
D2.1	Initial Visions, Scenarios and Use Cases		1.1	2018-06-04
D2.2	Initial reference architecture and PoC specifications		1.1	2018-11-26
D3.6	Final AI and ML features for smart behaviour and actuation		TBD	M32

2 Smart Behaviour definition and study

The first needed step is to establish a good conceptual foundation regarding Smart Behaviours. A brief review of Artificial Intelligence (AI) and Machine Learning (ML) concepts will help to describe what is exactly a *smart* component in the context of Brain-IoT architecture and to clarify where what is exactly a Smart Behaviour.

Once this is well defined, a small study and taxonomy of AI state of the art will help to point what kind of problems Smart Behaviours can solve, and what are the available techniques that make sense in the context of our BRAIN-IoT scenarios and how these techniques can be generalized into a coherent framework to align with the goals of interoperability and composability.

Finally, a study of both "classic" AI and ML, separately, will provided as basis for a first analysis whose goal is to discover which elements can be used to build a set of abstractions for a Smart Behaviours architecture that:

- Will be used to define a Smart Behaviours Application Programming Interface (API).
- This Smart Behaviours general design and API must be integrated within the Brain-IoT architecture (runtime environment, packaging).
- These elements will be managed from both the modelling tools and the IoT-ML language.

More focus will be put on ML methods, but this is only because recent years advancements in the field have turned techniques previously considered non-viable into very powerful tools, whereas agent systems or expert systems are already well known.

2.1 Context, definitions: AI and ML

The previously proposed definition for Smart Behaviours is: *the capability of autonomous decision and action taking in connected and data driven environments*.

In line with this definition, the kind of problems that Smart Behaviours must solve would be: reasoning, data analysis, forecasting, knowledge representation, perception or planning.

It quickly becomes evident that we are dealing with what is commonly known as Artificial Intelligence. But, due to the complexity and fast evolving nature of the field, some clarifications are needed about what is AI and ML, when is an algorithm considered AI, and what the general advantages and disadvantages of these advanced techniques.

Artificial Intelligence was defined by John McCarthy as *the science and engineering of making intelligent machines*:

"An attempt will be made to find how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves. … For the present purpose the artificial intelligence problem is taken to be that of making a machine behave in ways that would be called intelligent if a human were so behaving. (McCarthy, Minsky, Shannon, & Rochester, 1955)*"*.

This aligns with the intuitive concept that AI purpose is to study and build automated systems able to perform tasks that would normally require some degree human intelligence… But how is this behaviour achieved?

On one side we have a variety of tools that belong to "classic" AI, commonly referred as Symbolic Artificial Intelligence. These methods are based on high-level, human readable representation of problems, using logics and search algorithms. For these methods, the boundaries with traditional programming are not always clear. This is partly since some of these techniques have been partially incorporated to normal software development. A couple of examples of how sometimes the distinction between "normal" programming and Symbolic AI can be a blurred line:

- It can be argued that Object Oriented Programming shares the same foundational concepts (or at least a subset of them) with Frame Logic, which is considered a classic AI approach (Kifer, Lausen, & Wu, 1995).
- At its core an Expert System can be reduced to a set of if-then-else rules.

On the other side we have ML methods, which are dynamic in nature, adjust themselves iteratively in response to the data they are exposed to, and their design and development seem to differ greatly from classic development.

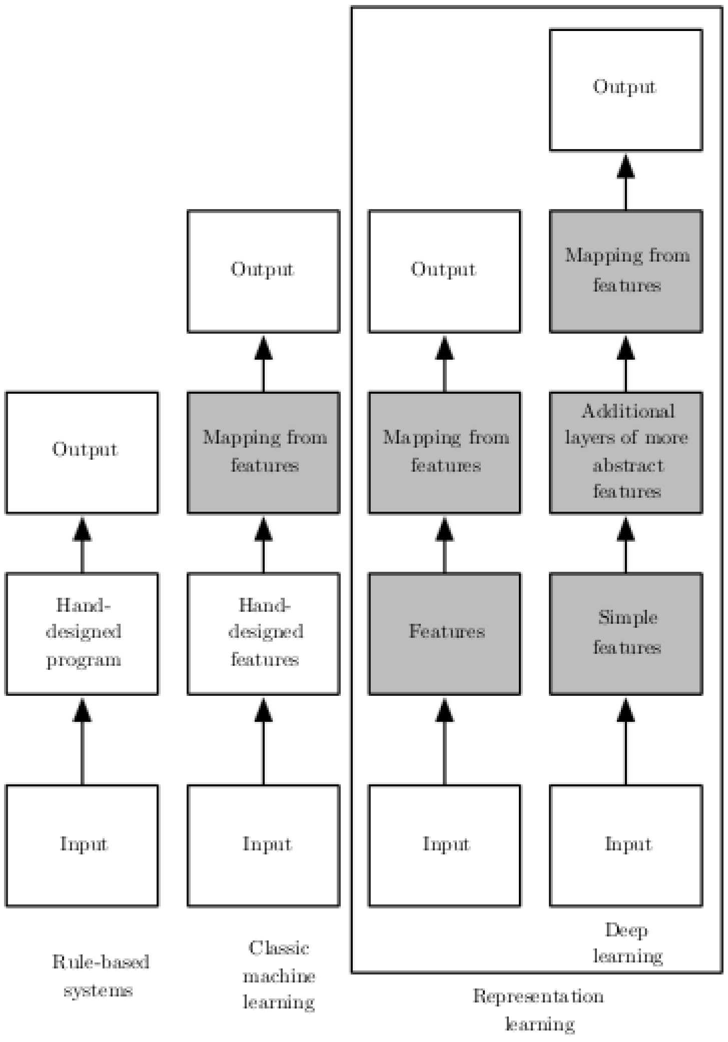

Figure 1: Rule-based Symbolic AI systems, "classic" ML, Deep Learning

<u>Machine Learning is a subset of Artificial Intelligence</u>, as show in Figure 1. A ML algorithm builds a *model* (this term has a specific meaning within the field) that, whatever its internal representation, behaves like a function f(x) = y that maps inputs to outputs. This ML model is refined and optimized with large amounts of

training data so after training the calculated values for outputs are reasonably close to either expected or valid/useful values.

It is important to point the difference between ML techniques and other more classic AI approaches. ML methods are well suited for:

- Problems for which existing solutions require a lot of tuning and a too big/complex set of rules.
- Fluctuating environments. An ML algorithm can adapt dynamically and without human intervention when new sub-behaviours are detected in the given inputs, whereas with traditional methods a new set of rules should be added to cover those new sub-behaviours.
- Problems for which the algorithmic/symbolic AI is too complex or even unfeasible.
- Scenarios where the goal is to gain some insight and new knowledge for a problem for which we have only large quantities of data (but don't understand its underlying structure or the relationships between its components).

All ML methods need a training phase, generally with human intervention and supervision, that is context and problem dependent. During this training phase some parameters that govern the ML model behaviour, and the ML model structure itself, will be modified in order to achieve better results.

2.2 Classic AI and smart agent system approach

While the task of generalizing Symbolic AI methods into a coherent view and set of common abstractions has a great complexity, and there are not libraries and frameworks that try to unify a set of different implementations under a common "Symbolic AI API" (whereas in the ML field, we do have such a thing), conceptually this topic has been already explored, notably Russell & Norvig famous textbook *Artificial Intelligence: A Modern Approach* (Russel & Norvig).

2.2.1 Autonomous Intelligent Agent

At its simplest form, an Intelligent Agent is a software agent (an autonomous program that performs a task (Nwana, 1996)) that perceives the current state of the world and applies some intelligence to this perception to choose an outcome.

It is assumed that an Intelligent Agent has both sensors and actuators to perceive and act on the environment. Although this aligns with some of the Brain-IoT use cases, we intend for our Autonomous Intelligent Agent architecture to be a more general abstraction, where inputs/percepts are mapped to outputs/actions and this mapping process can be any "classic" AI method.

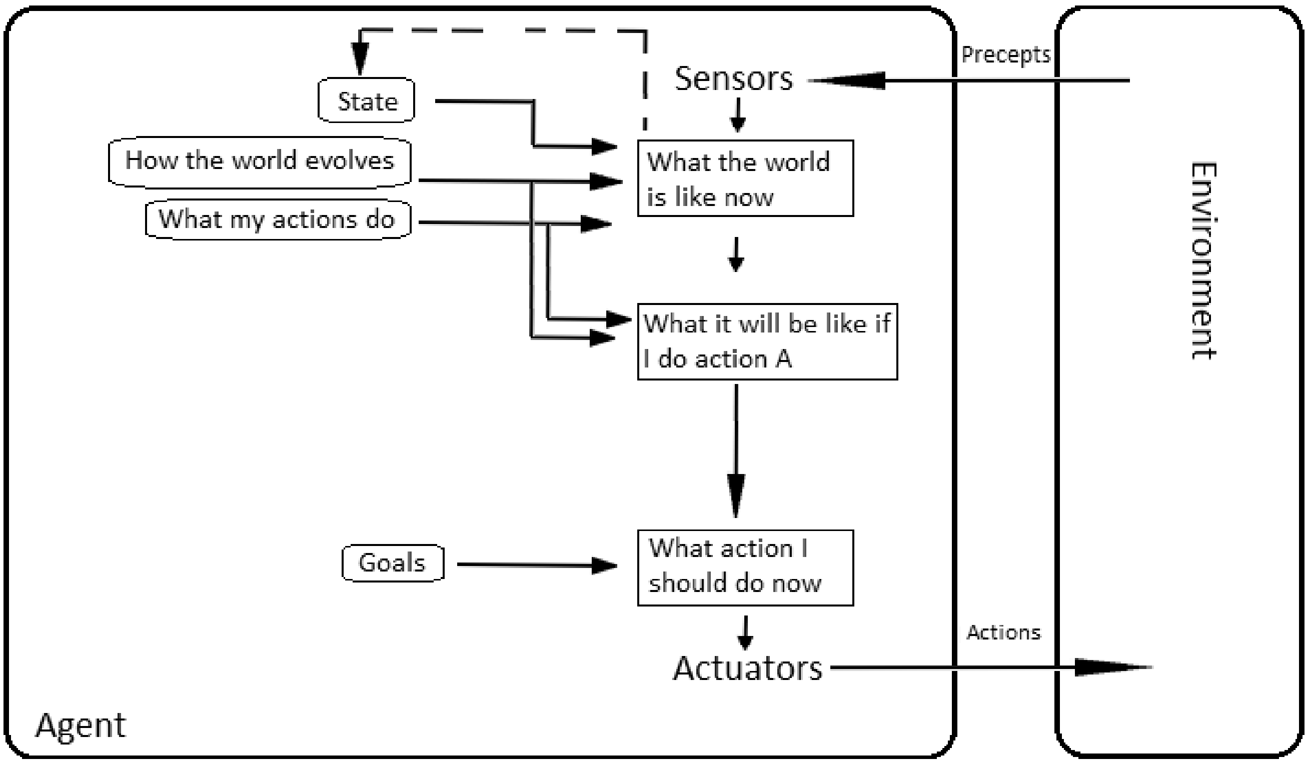

Figure 2: Example of an Intelligent Agent System design

The structure, design and complexity of the function by which the Intelligent Agent maps percepts to outcomes is then abstracted in this simple architecture, one possibility depicted in Figure 2.

The agent can be very simple in its decision process (a collection of if-then-else rules) or use some knowledge representation to store the current state of the world and then apply some inference process to that knowledge in conjunction with the inputs to obtain an output.

It must be noted also that the Intelligent Agent approach provides the bases for cooperative intelligent behaviour, exercised by groups of agents. This is the field of Multi-Agent Systems (MAS), where communication and coordination aspects within agents must be also managed. In this scenario, on top of individual Intelligent Agent considerations, we also have:

- Interaction over a network that will lead to partial failures.
- Incremental updates to individual behaviour that may also result in unexpected aggregate results, depending on the method/paradigm being used.

To solve this and other aspects, an orchestation is needed and we consider the approach developed in BIP could be very valuable for this kind of system where a group of Intelligent Agents must cooperate.

2.2.2 Types of knowledge representation and inference

Since the introduction of Expert Systems (computer system that emulates the decision-making process of a human expert with a set or if-then rules (Tan, 2017)), knowledge representation has been a key topic in the AI field. In general, any knowledge representation formalism uses symbolic logic to represent sentences in such a way that they can be used for automatic inference, yet still be human understandable.

A set of sentences meant to represent knowledge in a formal way form a knowledge base.

There is a great variety in complexity and human readability of the used logic formalisms to represent knowledge, but we can summarize the most important branches:

- Simple if-then rules (a subset of propositional logic).
- Frame logic, description logics, first order logic.
- Fuzzy logic.
- States of sentences linked in tree or graph structures.

Current advancements in semantic nets, and ontologies are always based in one of these formalisms. For example:

- OWL language, meant to integrate information over the web and allow reasoning (OWL Working Group, 2012), and built on top of RDF which itself is a language consisting in subject-predicate-object sentences but based on description logics (RDF Working Group, 2014)).
- WSML, a language that would be used to describe semantic services, is based on description logics and first order logic but has a syntax in the style of frame logic (Brujin & Lausen, 2005).
- Prolog, the famous logic programming language used for pattern matching over natural language, expert systems, planning. Rooted in first order logic (Prolog/Introduction to logic, s.f.).

For any automatic reasoning technique that is based on logic (including if-then rules), there are two main methods of automatic reasoning:

- Forward chaining, where the repeated application of logic inference (modus ponens) extracts new sentences out of the existing knowledge base.
- Backward chaining, which is also based in logic inference but begins with a goal state and searches antecedents for it.

When knowledge is described in such a way that we have a set of linked states of the world forming a tree or a graph, search algorithms can be used to find optimal or desirable paths. For example:

- Depth-first, breath first.
- A*.
- Iterative deepening.
- Simulated annealing.

Usually this kind of search in a space of linked states is influenced by some function that calculates cost or energy for each state. This cost/energy function can make use of heuristics. A heuristic function provides, for a given current state, a ranking value for each possible path where the calculated output may not be always the best, but it is considered good enough and can be obtained with reasonable computational resources (time and space).

2.3 ML methods: a basic useful taxonomy

A commonly used taxonomy for ML methods is to distinguish the different ways an ML algorithm can model a problem based on its interaction with the input data. This taxonomy is useful as an introduction that will underline the importance of input data processing and the model preparation.

Supervised learning
A model is designed, parametrized and then trained through an automated training process where its predictions are refined until the model reaches a desired level of accuracy that can be measured in a consistent way.

Input data is labelled, which means there is a training set of input data where, for each input value its expected result is known and provided to the ML algorithm. Training data must share the same type of input data.

The produced output are predictions about the results: the goal is either to calculate specific values (regression), or to infer to which category belongs each input data (classification).

Some relevant supervised learning example algorithms: logistic regression, Support Vector Machines (SVMs), decision trees, Naive Bayes, ensemble algorithms, neural networks.

In Brain-IoT these methods could be used for: data analysis, anomalies detection, decision making, time series forecasting.

Unsupervised learning
Input data is not labelled: the ML method is not provided with expected results for the given inputs to train. A model is trained generating structures and showing relationships existing in the input data.

Typical problems where unsupervised learning approach makes sense or is needed:

- Clustering to detect groups or categories within the given data, visualization to help understand data.
- Dimensionality reduction to simplify input data without losing too much information, anomaly detection, association rule learning.

Some relevant example algorithms of unsupervised learning: k-Means, LLE, Apriori, neural networks.

In Brain-IoT these methods could be used for: data analysis, image analysis, decision making (unlike in supervised learning approach, decision making here would require that the actions the system decides are evaluated later).

Reinforcement learning
This approach is based on the following concept: an agent senses its environment, selects an action, performs it and is rewarded (positively or negatively) in return so it can learn by itself the best strategy over time.

Known fields where this kind of technique is applied are robot movements and games like the famous AlphaGo Zero (Silver, 2017). This technique benefits from problems that can be reproduced in simulators. Great results for some real-world scenarios can be obtained combining deep learning with reinforcement algorithms.

In Brain-IoT these methods could be used for: industrial and/or robotic automation, decision making in high dimensional control problems, detection of anomalies in time series, machine tuning.

2.4 Considerations about Artificial Neural Networks and Deep Learning state of the art

Artificial Neural Networks (ANN) are a set of algorithms, or rather a framework enabling algorithms loosely inspired in our knowledge about the human brain, that are capable to recognize patterns in numerical data.

Although in its simplest form, a Perceptron Neural Network is just a classification tool, ANN can be used for clustering, classification or regression.

Deep Learning Neural Networks (DLNN) are a modern refinement of ANN that is being used in quite spectacular success stories, in such a way that today it appears to be the only form of AI worth practising. It is then a good idea to briefly review the main concepts behind ANN to understand the keys behind DLNN successful use cases but also clarify and underline their limitations.

At its core, a classic ANN is a set of interconnected layers of nodes. Each node processes a weighted sum of inputs through an activation function. This ANN must be trained with a set of inputs for which the predictions can be measured. The difference between the network guess and the ground truth or expected result is an error value that is used to update the model by adjusting weights (adjusting for each weight to the extent that the weight contributed to the calculated output).

A Deep Learning Neural Network is a neural network composed by at least three connected layers. Each subsequent layer can deal with more complex features, aggregating and recombining features detected in previous layers using nonlinear activation functions. Thanks to this, these neural networks can discover structures in unlabelled highly unstructured data without human intervention.

Deep Learning Neural Networks perform automatic feature extraction, and this alone makes them different to other known ML techniques. In "classic" non-NNs Machine Learning, the model needs an entry software layer that performs feature extraction: the provided inputs must be translated to a non-redundant set of derived values selected by the designers and developers using domain dependant knowledge.

2.4.1 Brief introduction to Deep Neural Network architectures

Four major architectures of deep networks can be distinguished:

- Unsupervised Pretrained Networks can learn a representation or encoding from a given unlabelled set of data, allowing for unsupervised learning. Here we have approaches like Autoencoders (that can be used for data generation or dimensionality reduction), Deep Belief Networks and Generative Adversarial Networks (used for synthetic data augmentation and generation).
- Convolutional Neural Networks (CNNs) can learn higher-order features in input data via convolutions. They are very successful in object recognition with images.
- Recurrent Neural Networks (RNN) model the time aspect of data by creating cycles in the network and are well suited for modelling functions for which the input and/or output is composed of vectors that involve a time dependency between the values.
- Recursive Neural Networks can also deal with variable length input like RNN but can model hierarchical structures in the training dataset, allowing to not only identify objects but also the relationships between them.

2.4.2 Deep Neural Network limitations

Deep Neural Networks have several disadvantages and limitations however:

- While it is true that they essentially perform automatic feature extraction, this happens in a way that is not human understandable at all. In come context, this can lead to legal issues, since predictions cannot be explained or justified in logical fashion. DL is non-explainable AI. By definition.
- DLNNs can produce astonishing outcomes in many scenarios already, but by their very nature, their design and development imply a lot of trial an error.
- It cannot be stressed enough that DLNN need a huge amount of valid training input data. Although this requirement of training dataset size holds true in most ML methods, DLNNs techniques are even more demanding.

This last factor alone renders DLNN unusable in many scenarios, where enough labelled input data does not exist, and/or good quality synthetic input data cannot be generated. Automatic labelling is of course a field where great effort is being put.

2.5 Study of well-designed ML libraries

Some famous successful cases of ML libraries are briefly studied, in order to get some insight about their core API elements so these concepts can be used for the smart designing Smart Behaviour ML blocks.

2.5.1 API design good example: Scikit-Learn library

This well documented API (Scikit-Learn, s.f.) is remarkably well designed and can be used both as an example and as solid inspiration. In this API all objects that abstract an ML technique share a simple interface based on:

- **Estimators:** any object that can estimate some parameters based on a collection of data (*Dataset*). Must implement a *fit*() method that receives as parameter either one *Dataset*, or two *Datasets* for supervised learning cases (where the second *Dataset* contains the labels given for the training set).
- **Transformers:** a special case of estimator that can also transform a dataset. It must implement a *transform*() method that receives the dataset to transform and returns the transformed dataset.
- **Predictors:** another specialization of estimator that can make predictions over a given dataset. It implements a *predict*() method that receives a dataset of new inputs and return a dataset with their corresponding predictions. It also implements a *score*() method that measures the quality of the predictions against a given set (and a second dataset of labels if it is the case of a supervised learning algorithm).

Every *Estimator* and all learned parameters are accessible directly in run-time. *Datasets* are expected to be arrays or matrices and parameters are expected to be Strings or numeric datatypes.

Composition of *Estimators* is encouraged. Sensible default parameters are provided with each object.

2.5.2 API design good example: Spark MLlib

Inspired by the already mentioned Scikit-Learn API, Spark MLib (Spark MLib, s.f.) also offers a unique standard API that makes it easy to combine multiple ML algorithms, where the central concept is the pipeline:

- **DataFrame:** a distributed collection of data (*Dataset*), organized into named columns. Originally a Spark abstraction equivalent to relational database tables or data frame as they are used in R or Python, ready for optimizations.
- **Transformer:** an algorithm that performs transformations on a *DataFrame*. It implements a method *transform*() that receives a *DataFrame* and returns a transformed *DataFrame*.
- **Estimator:** a learning algorithm that implements a *fit*() method which receives a DataFrame and generates a learned model that is a *Transformer*.
- **Pipeline:** a chain of multiple *Tranformers* and *Estimators* that defines a workflow as a sequence of *PipelineStates* that must be run in a specific order.

Parameter: the common abstraction for parameters used by all *Transformers* and *Estimators*.

3 Brain-IoT Use Cases analysis for Smart Behaviours

All the use cases where there are features that clearly to require some Smart Behaviour are briefly reviewed. AI/ML techniques that could solve or improve current scenarios are suggested for each case. Also, the current development status is shown.

3.1 Water Critical Infrastructure use cases and scenarios

UC-01.01 (Catchment optimization by consumption levels)
In this scenario it is described how the deposits shall be full between a defined range (two buoy values into the deposit) the most time possible in order to guarantee the water supply to the city. The entrance to the treatment plant (ETAP) has high limitations of capacity and the system infrastructure do not optimize the water discarded. Besides, the pumping has a strong inertia and its variation cannot be immediate.

The catchment system has two pumps, one of 500 litres/s and another of 1000 l/s that they take raw water from the river and delivered to the treatment process, and they pump water depending of the need of the headstock deposit. After the water treatment, there are four pumps that pump the potable water to the headstock deposit. These pumps are adjustable also in function of the need, but as the pumping of the raw water have only three possibilities, 500 l/s, 1000 l/s and 1500 l/s, sometimes it is necessary to discard water, with the power consumption associated.

The objective here is to make a prediction about optimal rate of pumping needed, using as inputs: the water consumption levels (current and historic), power supply fees and other factors. Techniques for time series analysis and forecasting could be employed.

Currently, there are rules based on timetable of peak of water consumption and timetable for power supply lower price, typically circa 8:00 (peak of water consumption), 14:00 (peak of water consumption) and 22:00 (advantage of the lower price of the power supply). There is already some kind of expert system in production environment. Some further analysis seems necessary.

UC-01.02 (Provisional monitoring systems)
In some situations, it is necessary to install provisional monitoring systems into the EMALCSA infrastructure (network or plants) in order to gather information from the environment. Currently there are systems to measure output water flow rate and EMALCSA is planning to install several rain gauges in some deposits of the water infrastructure. The rain gauges will be connected to an IoT remote station and it will collect the data of the pluviometry. The remote stations will transmit the information to the IoT platform SICA. This scenario represents a rain gauge network connected with SICA in order to retrieve the information of precipitations in the deposit localization.

The use case is more focused on "Thing" definitions and IoT services communication, although some smart behaviour must be built on top of the data provided by these monitoring devices.

As a suggestion, although it was not initially included in this use case definition, some smart behaviour concerning information forecasting could be used to help calibrating pumping, when consumption data is also available.

UC-01.03 (Verification of existing measurements)
The scenario described in this use case explains that after the installation and calibration of probes, the data can be considered correct, but when some probes and sensors are installed into the infrastructure, some readings can be strange, totally or partially. Sometimes data presents values out of range. These values

should be detected so to point possible device failures. And should be filtered as well, to keep data coherency.

Another behaviour regarding this scenario is that a device can be measure values into the right range of values, but perhaps it can be a wrong value because the devices are giving a right quantitative value but really the measurement is wrong. If some alert is triggered, we do not have the confidence if the alert is correct or if there is a problem with the device or the measures. Data analysis could be done to clarify what can be done in this scenario.

Based on this scenario there are is a clear first goal for a Smart Behaviour in this scenario: the detection of outliers in input data provided by monitoring devices. This could be approached from several techniques: clustering, SVMs, ensemble algorithms and Recurrent Neural Networks (RNNs) or Long Short-Term Memory networks (LSTMs).

UC-01.04 (Industrial security and safety of edge IoT devices in the infrastructure)
The current IoT critical infrastructure system is working with sensors and control systems non-IoT that connect with the SICA platform. These devices are in the edge.

It is planned to use this use case to test the security topics regarding the project, and in order to do that, the sensors, Smart Terminal and the Smart Gateway will be provided by Airbus to validate the security requirements and features. A MisNET sensor and gateway will be used for collecting the data and send them to SICA. Easiest solution to integrate MisNET and SICA would be to send data through sensiNact.

The final objective is to validate the security requirements and objectives described in the deliverable "***D5.1 - Initial Threat modelling and Security assessment***", with the goal of guaranty the security and safety of the scenarios where is needed to install measurement IoT devices into the critical infrastructure. The threats and vulnerabilities are defined in the first version of this deliverable.

As of this moment, no Smart Behaviour has been proposed to help with this use case.

This use case has been selected as part of first EMALCSA proof of concept. It will validate both device integration and industrial security & safety.

UC-01.05 (Reading process of water meters)
This use case deals really with the integration of different platforms that manage water maters. A crosscutting concern with smart behaviours is the definition of IoT-ML characteristics regarding AI-ML elements.

In this scenario EMALCSA has about 2000 water meters tele-managed with four monitoring technologies, but these systems are not connected with SICA or another IoT platform, so a goal would be guaranteeing the communication between these different platforms for exchanging data and information with a unique management platform in a strong and secure way.

This use case has been selected as part of first EMALCSA proof of concept, to validate platform integration.

UC-01.06 (Dam doors opening and water level analysis)
The dam of Cecebre provides water to the metropolitan area of A Coruña. When the level of water reaches a specific level, the gates or the dam valves shall be open, and the dam provides water to the river. This opening is triggered manually.

The goal in this scenario would be to provide some analysis regarding water level considering meteorological prevision, current water level and other factors like the water provided by the Mero and Barcés rivers. This scenario could be approached as a classification problem, but RNNs/LSTMs for time series forecasting could prove useful too.

This use case has been selected as part of first EMALCSA proof of concept. It will be used to validate Smart Behaviours integration with the Brain-IoT architecture, modelling tools and IoT-ML.

UC-01.07 (Control system for the water supply between deposits)
The goal for this use case would be to develop a smart control for the valve in a new Alvedro deposit, considering predictive algorithms in functions of consumptions and overall improving the control network.

A restriction would be that the water level into the Eirís deposit remains inside a defined range, so two main problems can be prevented: water overflowing the manholes and tube burst. A detailed analysis of these problems and other restrictions is provided in the expanded version of the use cases.

In the same way that UC-01.01 was analysed, this scenario suggests a mix of expert system with, for example, Boltzmann machines to infer parameters.

First Proof of Concept Scenario
The selected use cases for a first PoC are: UC-01.04, UC01.05, UC01.06. These will validate requirements regarding: device integration, industrial safety and security, platform integration and Smart Behaviours development integration.

As stated in previous point, UC01.06 contains a ML based Smart Behaviour. For this Smart Behaviour, a full initial analysis has already been completed. Current state:

- Good quality input data has already been provided by EMALCSA (covering water height and volume, precipitations and water flow provided, daily, since the 1980).
- A dataset has already been prepared and is being used to train ML models. A first simple version uses water volume and precipitations as inputs and water flow as output to predict.

Future iterations:

1. Add more inputs (other meteorological information to begin with) to refine predictions.
2. Use quality of water measurements, try to optimize also how the water flow is distributed between the different gates and valves.

3.2 Service Robotics use cases and scenarios

UC-03.01 (Simple pickup request)
The smart behaviours of this use case do rely on computer vision problems: detecting the load, using the actuators to pick it, detecting and avoiding obstacles, detecting the end point or worker... Also, we have classic search problem to infer a path for the robot, knowing the map of the surroundings prior to execution.

UC-03.02 (Collaborative pickup request)
This includes UC-03.01. The specific issues of this use case involve finding a set of available robots that can solve the task by collaboration. This seems like a classic search problem, maybe with some swarm optimization heuristic if multiple similar problems must be planned with a big set of robot candidates where each robot has different conditions and world state definitions. On top of that, we must deal with the communication and coordination between agents and the central unit. Potentially this is a Multi-Agent Systems problem.

UC-04.01 (Assisting another robot)
Behaviours included in UC-03.02. Communications and control needed in order to enable Smart Behaviour.

UC-04.02 (Sharing situation awareness)
Functionality needed by UC-03.X: how to detect and deal with obstacles that appear in real time. Communication protocols and policies about how to deal with a state of the world that is built and shared between the robots/agents. This seems also a Multi-Agent System kind of problem.

UC-05.01 (Actuation request)
This use case is needed by UC-03.01 and UC-03.02. The tasks involved would be: actuators design and integration, communication protocols and data formats definition.
The topology of actuators should be used to analyse and design set of actions the robot/agent can perform in order to solve the search problem (build a path to reach the objective).

UC-06.01 (Sensors feedback)
Independent sensors capable of detecting and classifying are very useful in an industrial environment. Crossing detection lines, bumpers, RFID tags, limit switches, barcode readers can be used together with a communication module to locate loads, exchange specific information or trigger a collaboration between machines (conveyors-robots, charging-stations, etc).

The platform will provide tools to model the infrastructure, including its related behaviours. Brain IoT platform will provide common framework for exchanging this information, trigger new behaviours and monitoring the status of the system.

This use case is needed by UC-03.01 and UC-03.02. Features to achieve: sensors design, integration, protocols for communication between devices and with a central coordination unit, data formats definition.

First Proof of Concept and new Use Case
A new use case has been proposed for the first Proof of Concept (PoC), in such a way that all core Brain-IoT components will be needed (core architecture, integration with an external IoT platform). And, at least, one non-ML based Smart Behaviour will be used.

In this Use Case, a fleet of N robots must pick up items of M different types in a designated pickup area and carry each item to one of M different storage areas (1 area per type of item). The layout is known in design time and there are dynamic elements that will require some interaction/communication in runtime, like a "smart" door.

The main purpose for a Smart Behaviour would be to optimize the set of routes needed for a fleet of robots to carry a given set of items.

Suggested methods: techniques for optimization like simulated annealing and genetic algorithms.

Current state: a simulation environment is being developed using Gazebo. Analysis regarding Smart Behaviour needs some refinement. The methods for routes optimization can probably be explored without any simulation environment at all. Interaction with elements in "external" systems (door, elevator, etc) can be developed independently.

4 Solution Manager System

4.1 Analysis to extract AI and ML behaviours from current BRAIN-IoT Use Cases

Our starting point will be to summarize the different kinds of smart behaviour blocks that involve either classic AI, normal ML or deep learning methods, and make sense to solve some of the features in the proposed scenarios. Both for the Water Critical Infrastructure and Service Robotics use cases.

4.1.1 Machine Learning behaviours

Neural Networks for classification
Classification is the process of predicting the class or category for a given input. Use cases like UC-01.06 seem classification problems that can be solved either with "classic" ML methods or using Neural Networks.

A sensible plan could be to build both models: as an exploratory tool, decision trees (introduced earlier) is a method that could show good enough results and has the advantage that it produces an output which is human understandable. A simple Neural Network (NN) model can also be trained if/when there is enough data to provide a useful input dataset.

Restricted Boltzmann Machines to infer missing data
Restricted Boltzmann Machines (RBM) is a method that allows finding patterns in given data reconstructing the inputs. Consisting in a two layers neural network where each node of the visible layer is connected to each node of the hidden layer, there is a unique bias parameter that is used to calculate the activation outputs in a forward pass and reconstruct the inputs in a backward pass. Another parameter called KL divergence is used to compare actual input values with recreated ones.

This technique became famous for its contribution enabling current Deep Learning (as it helps solving the "vanishing gradient problem"), but it can be used also for clustering and infer missing data in given datasets.

RBMs then could be used to assist in use cases like UC-01.03 and any other use cases in which a complex not 100% complete dataset must be used (for example UC-03.06).

RNN for time series forecasting
One property of traditional NNs and modern Convolutional Neural Networks is that they lack memory. Each input shown to them is processed independently, with no state kept in between inputs. With such networks, to process a sequence or a temporal series of data points, it is needed to use the entire sequence as one single input for the NN.

Recurrent Neural Networks are inspired by the way biological intelligence processes information: incrementally, while maintaining an internal model of what is being learned. RNNs have loops that allow information to be persisted.

Long Short-Term Memory networks (LSTM) is a type of RNN that deals specially well with long-term dependencies in the input data. This method is very successful dealing with multidimensional time series forecasting and can be employed in use cases UC-01.01 (levels of water consumption), UC-01.03 (detection of outliers in sensor outputs), and UC-03.06 (dam doors opening).

4.1.2 AI (non-ML) behaviours

The main task an agent of UC-03.01 must quickly resembles a classic computer science problem, which is basically finding the optimal path and/or series of actions needed to achieve a goal point or state, given an initial point and/or state of the world.

This is a problem that can be solved with very well-known algorithms that don't involve learning, like Dijkstra, iterative deepening or A*. These algorithms need a precise quantitative description of the world state to be available, so they can compute a "cost" required to perform a step to extend the path towards the goal state.

The Service Robotic scenarios involve also object detection problems. These can be solved using quite different computer vision methods and approaches. Some of them involve ML (CNNs, Yolo) but there are also effective ways to perform simple object detection with simpler techniques.

To deal with human operators, objects to carry and areas where the objects are to be deployed, classic non-ML methods aided using physical markers can be useful and very practical. To deal with obstacles in real time and other aspects that must be considered in real world scenarios, an approach that involves DLNNs trained with useful datasets… adapted to robots point of view and constraints, something which may require a lot of work building a good enough dataset.

The use case added for the Service Robotic first proof of concept also includes a planning/optimization sort of problem. The goal is to choose optimal sets of routes for a fleet of robots that must reallocate items traversing a known area. This use case implies making use of the functionalities developed for the Service Robotic use cases.

This could be solved with techniques like simulated annealing, search algorithms with heuristics, maybe genetic algorithms.

Suggested elements for the modelling and formal description of Smart Behaviours: search algorithm, function cost, heuristic cost, world state description, agent, agent actions, marker, physical objects classification, object detection CNNs based algorithm, object detection training set, object detection AI algorithm, routes optimization AI algorithm.

4.1.3 Reinforcement Learning behaviours

UC-03.01 could also be solved, or refined, using Reinforcement Learning. This approach has similarities with classic search: the agent must know the sequence of world states and actions. But here, the agent's behaviour seeks to maximize the expected cumulative reward. With these methods, some problems the robot may encounter in real-time scenarios can we "rewarded" with negative values, so the actions chosen to advance through the path are refined in next iteration.

Classic Reinforcement Learning uses algorithms like Monte Carlo and Temporal Difference. Q-Learning is a method that uses a sort of table that is used to compute the future reward of an action given a current state.

With deep learning, a technique known as Deep Q-Learning can provide good results for problems where classic Q-Learning could not be used due to the size of the state space.

Suggested elements for the modelling and formal description of Smart Behaviours: agent, world state description, agent actions, reward function, deep Q-Learning table and model.

4.1.4 Agent collaboration behaviours and other considerations

Use case UC-03.02 seems a Multi-Agent System (MAS) kind of problem, although in MAS a central station is not needed, at least not for coordination (although it can make sense for communication purposes, as gateway, etc).

This paradigm is based on autonomous agents (in this case, the robots) that can work alongside humans, where each agent shares its knowledge about the world state with the available agents sending messages in a language like ACL (used in JADE (JADE, s.f.) for example). An agent has intelligence, which may be AI like search algorithms or reinforcement learning (which may involve deep NNs or not).

Suggested elements for the modelling and formal description of Smart Behaviours: agent, agent actions, world state description, current world state description message, agents gateway, agents training.

4.2 Proposed design: overall view

Trying to generalize AI methods is an ambitious goal, due to the huge variety in available techniques available and the fact that most (if not all) of them are very context dependent.

To be able to build such a system, generic enough that it can include most of the currently known methods but also allowing for real-world implementations with all the fine-tuning detail so necessary for these techniques. It seems reasonable to subclass this system into two different sub-systems, as shown in Figure 3:

- Agent Solution Subsystem.
- Machine Learning Solution Subsystem.

Although both sub-systems share a similar structure and have almost equivalent components (which it is of course convenient), we think this distinction is needed because the focus in these two approaches differs greatly and it is the source of the division between Machine Learning and the rest of Artificial Intelligence:

- Symbolic AI algorithms are fully designed by their developers to encapsulate some intelligent decision on data whose relationships and dependencies must be known at design time.
- Machine Learning models are partly designed by developers, but their intelligence is acquired through a training stage with proper input data. This intelligence will rarely be something that can be transposed to human readable rules.

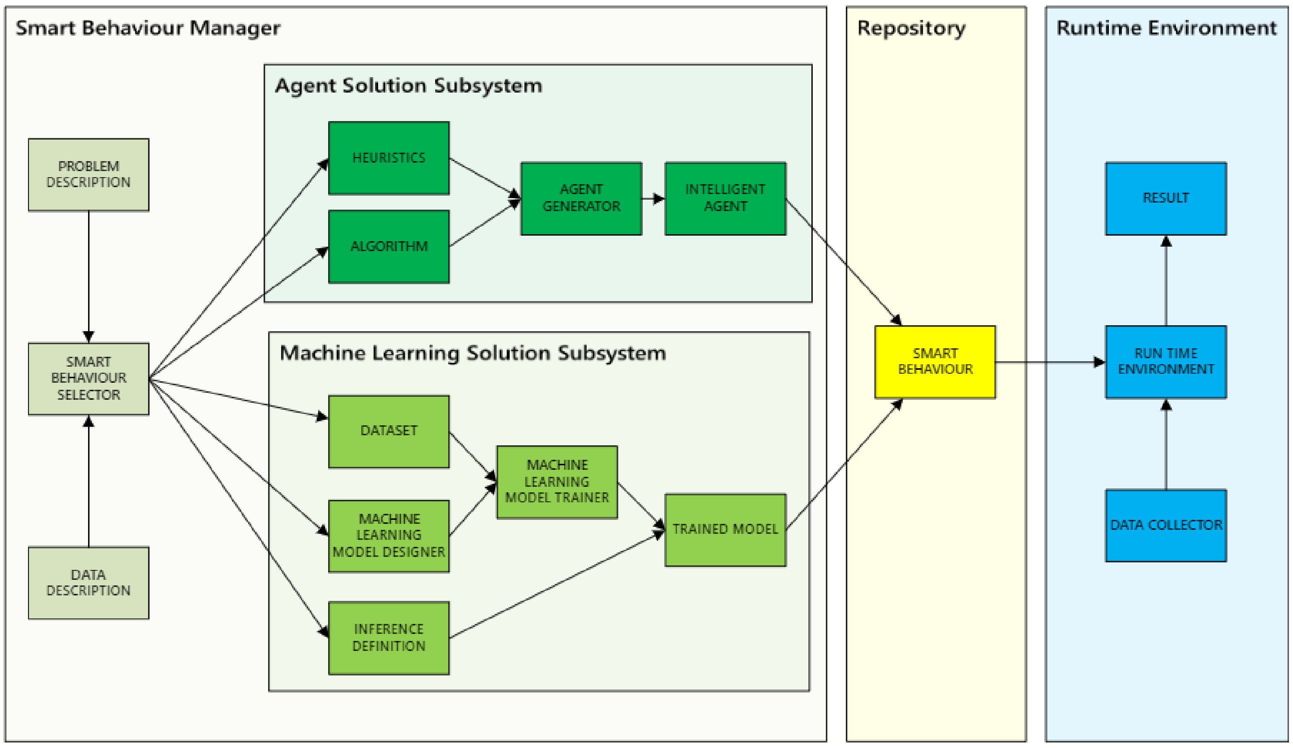

Figure 3: Smart Behaviour overall system, first proposed design

State of the art previously discussed supports this distinction: neural network libraries like Keras provide a widely used and quite high-level API and libraries like Scikit-learn provide an abstraction for most Machine Learning techniques. It is not easy to find such generalization with non-ML methods and the Smart Agent approach seems a valid design to abstract these.

Moreover, another fundamental distinction regarding coordination and orchestration can be pointed:

- Systems abstracted under the Smart Agent paradigm can be subject for coordination and/or collaboration dynamic rules.
- ML systems will generally have a more monolithic architecture, unless abstracted also under the Smart Agent paradigm in order to collaborate with other behaviours[1].

Common abstractions and basic blocks for a Smart Behaviour context description and problem definition

Semantic description of both the context of the domain and the type of problem that needs the collaboration of a Smart Behaviour is needed, not only to inform developers and designers of the constraints that pushed for a solution, but also to be able to run matchmaking services with available Smart Behaviours in a marketplace or repository.

Data formats and restrictions for both inputs and outputs

For different reasons and under distinct approaches, both Symbolic AI and ML methods need a very precise specification of inputs and outputs. In the AI field, a description of the possible world states and transitions between them is mandatory. In ML models, input and output data must be an N-dimensional array or matrix, because most techniques use linear algebra as their basis for training and prediction operations.

Each Smart Behaviour implementation must then provide a detailed description of inputs and outputs. Specifying data format and training dataset restrictions along with domain specific rules (example: an ML model trained for face recognition using frontal face images will not be able to detect people seen from above).

Model definition: structure design and hyperparameters/heuristics

Under model definition, two different elements must be distinguished:

- The structure and design of the ML model or AI algorithm. This will very be specific for chosen method.
- The values used for "hyperparameters". This is a term commonly used in Neural Networks to refer to those parameters that are specific to the model to be trained and must be set before an automated training phase is executed. We are going to use this term to also abstract parameters that must be set for AI non-ML techniques, including the definition of heuristics.

There is a one-to-many relationship between model design and hyperparameters: for each model design, different values will be used, adjusting them iteratively with the goal to improve the obtained model's accuracy.

4.3 Decoupling inputs/outputs interface from behaviour implementation

In this first design for an abstraction that encompasses both subsystems as much as possible, to be able to provide a unique API, the following elements are presented:

[1] Not taking into consideration here other orthogonal architectural aspects within the ML system itself: distributed computing and persistence.

- A common set of data structures for both Inputs and Outputs. In the case of actions that cause change in the world state the agent is interacting with; the *Action* is supposed to inform about the effect that was produced by the agent.
- In the same manner, a common abstraction for all the *Configuration Parameters* ("hyperparameters") a behaviour algorithm may need.
- A (external) representation of the *Smart Agent* and its communication capabilities with other agents (choreography).
- An abstraction of a behaviour the agent knows and can provide, in the style of the Strategy Pattern (from GOF Design Patterns). This behaviour may be as simple as a set of rules or as complex as a DLNN, but it is invoked through a simple *process()* method that receives a *Dataset* and returns another *Dataset*.

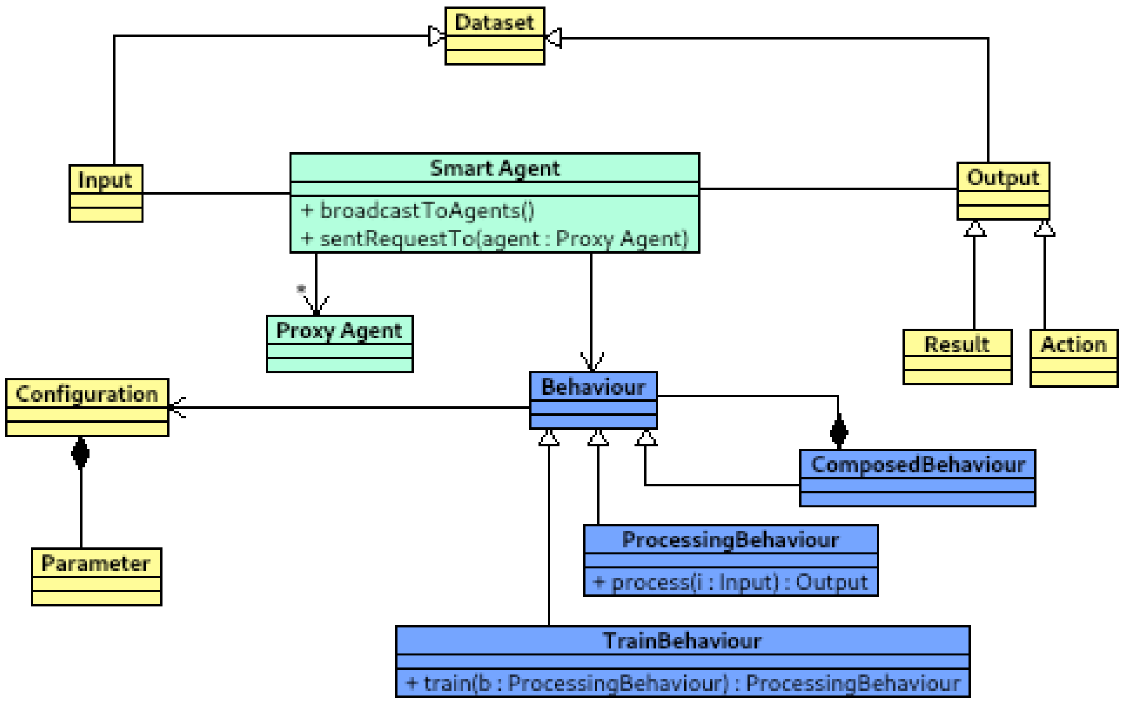

Figure 4: Smart Behaviour design for a set of high-level abstractions

The behaviour can be a composite (an ordered chain of behaviours). Also, some ML behaviours will require the execution of a *TrainBehaviour* until the model is prepared or sufficiently assessed.

4.4 Training and production environments, support for different lifecycles

As pointed in D2.2, different styles and types of development cycles regarding AI systems and ML models must be supported in BRAIN-IoT architecture and Smart Behaviours design and abstractions.

Two different dimensions with distinct possibilities and restrictions must be observed:

- In ML models and AI algorithms that need training that can be executed in a different environment, not directly connected to production environment (offline training), or it may be necessary/convenient to perform training to update an already deployed model.
- The predictions calculated by the Smart Behaviour can be batch processes, invoked on-demand, or needed in real time performed on streaming data.

It is unclear at this point how this crosscutting concern should be handled in Smart Behaviours modelling and specification. However, it will be a fundamental issue in Smart Behaviours development, and it should probably be included in their design and will be a key point in their deployment.

4.5 Machine Learning Solution Subsystem unique elements

There are elements specific to the Machine Learning Subsystem, that must be managed when dealing with the automatic learning process of the ML model.

Model learned parameters

In ML methods, the training phase will calculate values for the parameters used by the chosen and designed model (example: the weights of a Neural Network). Each combination of model and hyperparameters will produce one or more different learned sets of parameters.

Being able to keep track of the different sets of *models* – *hyperparameters* - *learned parameters* along with versioning and other metadata can be very important for the Smart Behaviour designers, developers and maintainers.

Valid training set and testing set (and/or evaluation method)

In ML supervised training the definition of both training set and testing is mandatory. The validity of the predictions for the trained model must be tested against a set of inputs. The testing set must be disjointed with the training set (if this were not the case there would be no way to determine whether the learning process is being successful or not).

Both datasets, training set and testing set, must share the same specified format.

4.6 Different treatment and requirements regarding datasets

Although our proposed design abstracts inputs and outputs for both subsystems, the significant differences in the structure of input and output data between Agent and Machine Learning approaches must be pointed.

By the nature of the Agent subsystem algorithms, the provided inputs must be structured in such a way that can be managed by the knowledge base of the Agent and used by its inference engine. In this way, for the Agent subsystem the translators/parsers must translate inputs and outputs to one of the formalisms mentioned in section 2.2 of this document. Any abstraction that tries to generalize these should probably be a graph structure of logical sentences, in the style of RDF.

The Machine Learning subsystem however will always need inputs to be translated to N-dimensional arrays (commonly 2 <= N <= 5). Fortunately, this datatype is already directly supported in any major DLNN and ML library and API (Dataset, Dataframe).

4.7 Assumed limitations, adopted conventions, pending work

Trying to design these abstractions so they can be used for the modelling, specification and development of Smart Behaviours, we opted to not include Machine Learning clustering or any purely unsupervised method. At least, not initially. In this way, we conveniently can abstract any other ML technique (either for classification or regression) with a simple interface.

Although not developed in this proposed design, reinforcement learning techniques could be abstracted with the Intelligent Agent approach. In those cases when reinforcement learning must use also Machine Learning, it must be defined whether either this can be supported by just a composition of two Smart Behaviours, or a way to integrate ML elements into the Intelligent Agent framework is needed.

As stated in our proposed initial design, ML based Smart Behaviours will tend to have a monolothic architecture regarding their functional/operational management (contrasting to the MAS viable within the Smart Intelligent Agent paradigm). However, it must be clarified that current implementations for real-time predictions (especially the ones dealing with Big Data) are already based per se on distributed highly optimized computation. And the implementation for this distributed algorithm execution is be handled entirely by the ML system, as it is the case for Spark (in-memory execution engine with clusterization that runs entirely in the JVM).

5 Design, modelling and formal description of Smart Behaviours

In D3.1, IoT-ML is presented as historically built as a UML-based language, but currently built upon IoT-A. Also, in D3.1 several UML behavioural diagrams are considered to represent all different levels and types of behaviours needed for the BRAIN-IoT Federations and Nodes.

Using the design proposed in previous section, the list of generalized elements for Smart Behaviours will be briefly reviewed, underlying what roles can each of them play in Smart Behaviours development and management. The purpose is to provide a basis for D3.3 and D3.4 regarding Smart Behaviours.

5.1 Smart Behaviour elements and their visibility

For a first approach, we are going to distinguish the following roles or actor views, regarding Smart Behaviour elements:

- Designer/developer/maintainer. Any human actor who has part if the development of a Smart Behaviour that is created from scratch. This role needs access to the whole picture, all the details of a Smart Behaviour (internals and relationships).
- Matchmaker. An automatic or semi-automatic process that searches for candidate Smart Behaviours. This role needs to handle metadata enough to enable automatic matchmaking where Smart Behaviours capabilities are compared with a set of given requirements.
- Composer. A human actor that has part if the development process of a Smart Behaviour, but not the entire workflow. The distinction here is that a composer will only reuse existing Smart Behaviours and Brain-IoT components and will not create a Smart Behaviour from scratch.

List of elements/blocks extracted from our proposed design for Smart Behaviours:

- Context description and problem definition (metadata).
- Dataset: inputs, outputs.
- Datasets formats and restrictions (metadata).
- Translators/parsers to transform between domain objects and datasets.
- Training set, testing set or evaluation method.
- Algorithm or model structure design with versioning.
- Algorithm or model "hyperparameters" with versioning.
- Model learned parameters (weights) with versioning.
- Communication elements for smart agents.
- Encapsulation or wrapper for the smart behaviour, that allows composability.

In the Table 1, visibility / access to Smart Behaviour elements for each role is depicted:

Elements	Development	Matchmaking	Composer
Context description	Yes	Yes	Yes
Problem definition	Yes	Yes	Yes
Dataset	Yes	No	No
Dataset format	Yes	No	No
Dataset restrictions	Yes	Yes	Yes
Translator/parser	Yes	Yes	Yes
Training set	Yes	No	No

Testing set / evaluation method	Yes	No	No
Model definition	Yes	Yes	Yes
Model structure	Yes	No	No
Model hyperparameters	Yes	No	No
Learned parameters	Yes	Yes	Yes
Smart Agents communication	Yes	No	Yes
Smart Behaviour composable wrapper	Yes	Yes	Yes

Table 1: Smart Behaviour elements and its use for each proposed role

At the current moment of Brain-IoT project, two PoCs are being developed. This is already providing useful feedback to clarify how Smart Behaviours should be handled by modelling tools and IoT-ML. The next step after this document is to provide a first precise definition for a subset of properties for each described Smart Behaviour element.

5.2 Some considerations about modelling, formal verification and Smart Behaviours

Brain-IoT modelling framework incorporates the features provided by BIP language, allowing to build complex systems by coordinating the behaviour of a set of atomic components. BIP is model based, component-based, and supports a rigorous design flow. Allowing for verification and real-time simulation using BIP engine.

One of the goals of BRAIN-IoT is to support these features in the development cycle of IoT systems. But Smart Behaviours present a difficulty, due to their non-deterministic behaviour. Tools like statistical model checking provided by BIP-SMC cannot be applied.

Following this reasoning, it has been decided to:

- Treat Smart Behaviours as "black box" components hidden behind an external simple API, as to be able to use the exogenous coordination features of BIP.
- Study other state-of-the-art different approaches as to how perform formal verification of ML based Smart.

5.3 ML Training to production workflow: a small topology

Two or three dimensions can be used to categorize the different ways in which machine learning models can be put into production environment: how/when are predictions made (forecast batch vs real-time on demand), how/when is the learning process taking place (static offline training vs dynamic training), and whether we need scalability for high throughput.

These dimensions give us different scenarios:

- Batch prediction, static training. A static dataset is used to run the model on it, a prediction is calculated offline and the results are submitted somewhere. A simple automation can be added to schedule a service to perform predictions periodically and write the outputs to repository.
- On-demand predictions, static training. The model is still trained offline, but a simple web service takes the input parameters (a single record instead of a dataset of entries) and returns the prediction for those inputs. This approach is quite common, using for example a Flask microserver to interface

with a trained model and built with Tensorflow (with or without Keras) or PyTorch. Other technologies like Azure ML offer features of this sort, but also allowing for a more complex data workflow. A commonly used way is also TensorFlow Serving, which basically offers a REST API too.

- Real time predictions, static training. With real-time streaming analytics, using Spark for example, a stream of events is used as input data and the model is always running on the data as it enters the system. This is very used for anomaly detection over large quantities of data, and it makes sense for systems when analysis over a lot of IoT devices must be performed.
- Real time predictions, dynamic training. It is already possible with technologies like Spark to keep improving and updating the model while in production.

Real time or on-demand predictions, automatic ML. The more sophisticated workflow would be to not just update the model in production, but to even produce entirely new models as the input events are processed.

6 Runtime Environment and integration with Brain-IoT architecture

A very brief recap of BRAIN-IoT architecture is presented, focusing on how the AI / ML elements of Smart Behaviours could fit and what restrictions and requirements must be observed.

The available and desirable technological options for both runtime implementation and model/data exchange formats regarding AI/ML elements of Smart Behaviours are considered and evaluated.

Finally, available options for Smart Behaviour packaging are also considered and evaluated, aligning with previous subsections. A proposed solution is presented.

6.1 BRAIN-IoT architecture introduction and considerations

In the BRAIN-IoT architecture, introduced in D2.2 and being refined as part of WP4:

- A BRAIN-IoT Federation is comprised of several BRAIN-IoT Fabrics/Members.
- Each BRAIN-IoT Fabric contains one or more BRAIN-IoT nodes.
- A BRAIN-IoT Node is an instance of an OSGi R7 complaint framework supporting one or more services and behaviours.
- All BRAIN-IoT artefacts must be accessed from an OSGi Alliance R7 compliant OBR repository.
- A BRAIN-IoT Edge Node is a type of Node that can interact with Physical Things, using dynamically loaded servients based on W3C WoT.

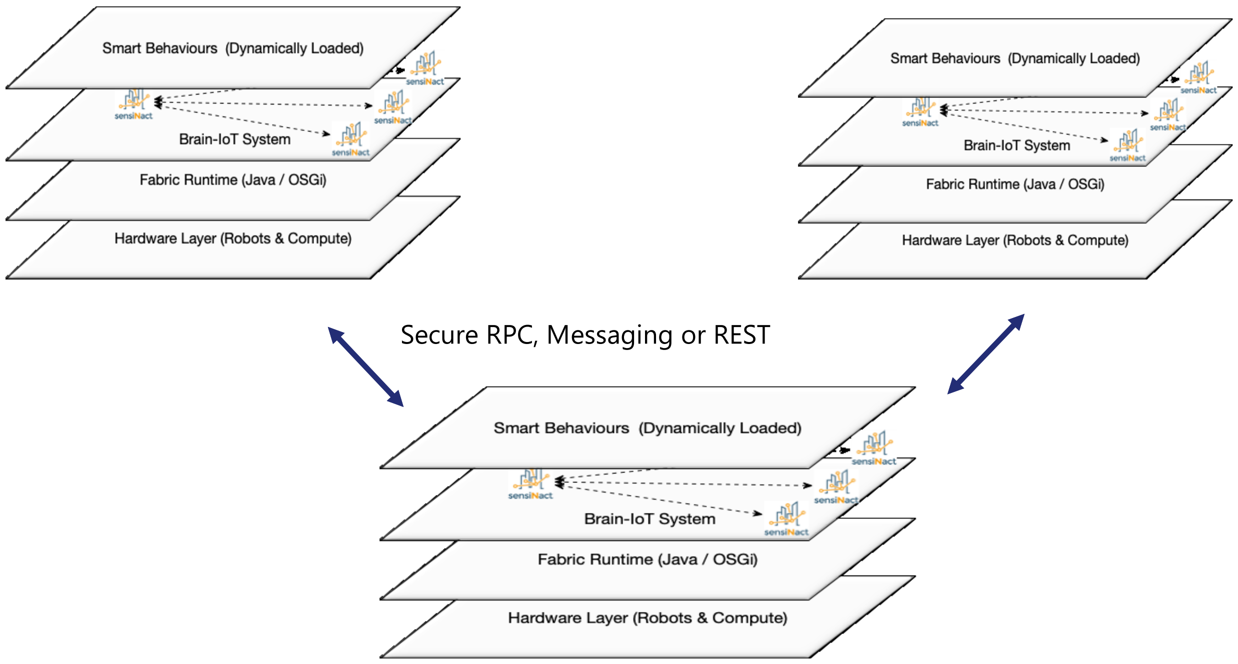
Smart Behaviours (Dynamically Loaded)
Brain-IoT System
sensiNact
Fabric Runtime (Java / OSGi)
Hardware Layer (Robots & Compute)
Secure RPC, Messaging or REST

Figure 5: Structural Federations with Smart Behaviours (from WP4)

From the technical design for this architecture it follows, as it is stated also in D2.2, that: "*a BRAIN-IoT node (…) can run on any platform that supports a Java 8 (or higher) JVM*". However, this constraint is being relaxed especially as the JVMN is becoming more modular since Java 9 and more even so since 11. Also, JVM/OSGi ability to assemble modular systems from self-describing components is one of the goals for BRAIN-IoT architecture.

6.2 AI / ML elements runtime implementation

As it was introduced in D2.2, many of the most popular available ML libraries and frameworks today offer Python APIs as a relevant key point for their widespread use.

Pure Python algorithm implementations are traditionally too slow to run in production environment. Traditionally development cycle for ML models would use first an environment based in Python, Matlab or R for exploration and testing, and then translate those models and/or algorithms to a C/C++ based environment for much better optimizations.

However, ML current ecosystem is different. Modern Graphics Processing Unit (GPU) is used for ML and provides a big differentiation factor allowing new architectures and implementations that were not feasible previously. In this way, several considerations are made here concerning Python ML stack as an example to explain and describe these changes:

- Well known for years libraries like Numpy or Skicit-Learn are heavily written in C.
- Still, it was common that, while operations in those libraries are transposed in C++ or Fortran libraries, data was being translated back and forth to Python variables.
- Current state-of-the-art frameworks to train and deploy ML models are wholly C/C++ and CUDA based implementations (Tensorflow, XGBoost, Torch…). And thin Python APIs or bindings are provided for the end users, which are either ML developers or data scientists. Tensorflow, very successful right now, is a good example: the graph (NN model) execution runs zero Python code, unless debugging is involved in the process. The Tensorflow/Pytorch/Theano libraries are the actual software components that perform the multivariate differentials and floating-point arithmetic.
- Python itself allows for huge optimizations using Cython (static types and compiling Python into C code) and Numba (JIT compilation using annotations, can make use of GPU for vectorized operations). Some benchmarks show very similar performance for C, Cython, Python using Numba compiler, and Julia language for operations with matrices (Puget, A Speed Comparison of C, Julia, Python, Numba and Cython on LU Factorization, s.f.).
- While it is always feasible to achieve better performance by rewriting model and algorithms to C/C++, it is not in any way nor easy neither cheap in terms of development effort.
- Processing huge datasets, a much bigger and relevant optimization is achieved using a cluster of machines with technologies like Spark.
- Generally, with the rise of GPU accelerated computing, used through libraries like cuDNN, more and more of the actual computing for ML is being offloaded to GPUs. By this factor alone any performance advantage C++ implementations still have is becoming almost irrelevant.

There is a reason for the rising popularity of R or Python in ML and data analysis fields (Voskoglou, s.f.) (Puget, The Most Popular Language For Machine Learning Is …, s.f.), and why technologies like Tensorflow and Torch with PyTorch offer Python bindings. Taking Python (including the data science libraries like Numpy, Pandas, Matpotlib, etc.) again as a case study:

- Python provides an accessible and high-level development environment for data scientists and non-software engineer machine learning experts that allows integration with a C/C++ backend.
- It is also a great option to write prototypes and modify them while debugging, something that is essential for the development of ML models.

6.2.1 Conclusions about implementation technologies for ML behaviours

Taking previous point as bases, we can offer some conclusions:

- There is not a single dominant technology stack or platform for ML. Python, R, Java, C/C++ and others have different strengths, sometimes overlapping.

- We have studied, as a good example of current tendencies, the rich and successful Python ML ecosystem. This is built on top of C/C++ libraries that make use of available GPU programming and CPU optimizations.
- Any case where production environment must deal with real time and/or big size datasets needs a different approach (in the style of Spark). This is partly due to inherent Python self-imposed limitations (no multi-threading allowed per se).
- It still makes sense to re-code in C++ an ML algorithm originally developed in Python, Matlab, etc. using the learned weights and parameters. But it is not needed as it was before the introduction of the current ecosystem and the popularization of GPU processing for ML.
- The development of full C/C++ ML solutions is a better option (or even the only option), either when the problem cannot be solved using common models and algorithms, or when it is necessary to port the implementation for embedded devices/platforms with computational and power usage restrictions.

As the already proposed architecture will host and execute the Smart Behaviours in the Edge devices or even at a higher level, and not in the Things themselves, this last point is not so relevant within the BRAIN-IoT scope and current goals.

We consider this current success these technologies like Python ML stack or R for scientific computation enjoy is not accidental and it will either continue in future time or be transferred to other options that provide similar characteristics.

This study has focused almost solely on ML technologies, but the same reasoning applies to the Symbolic AI field, specially that there cannot be a dominant technology stack platform. However, we can observe a difference. Lacking the very optimized widespread libraries and frameworks ML (specially DLNNs) developers enjoy, ad-hoc optimizations (resorting to C/C++ when necessary) are and will continue to be more common.

6.2.2 An additional thought about Machine Learning field

Current state of the art technologies did not even exist two or three years ago. Pytorch, being already quite popular (if not as much as Tensorflow), has its 1.0 version still released in 2018 offering full support of ONNX and integration with Caffe for production environments. Caffe 1.0 was released on 2017.

These ecosystems are changing very rapidly. It seems necessary for the Brain-IoT architecture to be prepared to support any evolution or new addition to these ecosystems.

6.3 AI / ML data exchange formats

Several options have been considered regarding the interchange of AI / ML "models". Some of them were already mentioned in D2.2. This is a brief analysis:

- ONNX (ONNX, s.f.). Allows to save Neural Networks models. Direct support for CNTK, PyTorch and Caffe, There are also external converters for Tensorflow and Keras.
- NNEF (NNEF, s.f.). Neural Network models interchange. Maybe a more ambitious design than ONNX. Similar support.
- PMML (PMML 4.3 - General Structure, s.f.). Interchange mark-up language for predictive models, with support for Neural Networks and other ML methods like SVMs, decision trees, etc.

A Neural Network data exchange format allows neural network graph data to be "exported from" and "imported into" an environment. The use of a Neural Network data exchange format allows the training runtime to be decoupled from the destination Production runtime environment: e.g. a Neural Network may be trained in MatLab, and then exported via a neutral data format to be run in production in a Java, C or Python environment.

Several considerations must be observed:

- While ongoing standards like ONNX and NNEF look good enough, their support for the different Deep Learning Neural Networks technology stacks is not complete. And this is partly due to political factors that may not be expected to be solved.
- As previously stated, ML field is quickly evolving, and it seems wise to not depend on ONE existing interchange format.
- Only a subset of Smart Behaviours can benefit from this feature and use an existing viable interchange format. PMML covers a broader range of ML techniques and "classic" Symbolic AI methods lack a developed interchange format standard.
- Whichever approach is chosen by BRAIN-IoT, the release artefact (interchange neural network format) will be expressed as an OSGi bundle, with appropriate Requirements & Capabilities metadata. Some OSGi Alliance Specification work may be appropriate here.

6.4 Smart Behaviour packaging: options and proposed solution

As stated in previous subsection, it is not feasible to decouple in all cases the AI "model" from the other subcomponents (communication, external API). In view of this, at least one mechanism able to package and deploy the whole Smart Behaviour is needed.

Several options have been considered as to how Smart Behaviours should be packaged.

6.4.1 Option: a packager deployed opaque artefact.

A valid option would be an OSGi bundle with appropriate coarse-grained Requirements/Capabilities metadata, deployed to native OS or local docker container.

- This would give support to any implementation for a Smart Behaviour.
- In this way, lifecycle and configuration would be controlled by Edge Node.
- It would be needed to support management for different Packager bundles, deployed dependent on underlying hardware "Capability".
- Communication mechanisms between AI behaviour and sensiNact Edge Node to be determined.
- Internal dependencies are not managed.

6.4.2 Option: a packager deployed `assembly` for Python and C++ modules

Another option would be to use these frameworks that provide an OSGi 'like' modular environment for Python or C++: iPOPO (What is iPOPO, s.f.) and C++ Micro Services (CppMicroServices, s.f.).

- However, both lack the Requirements / Capabilities model or the framework resolver.
- A new bundle must be loaded by an external actor - dependencies must be explicitly managed by the external actor.
- At runtime these approaches probably provide little / no Operational advantage over the opaque artefact - though promote mode modular code.

6.4.3 Option: deployment of a Java / OSGi assembly

If we restrict AI and ML implementations to the JVM, it could be possible to handle the whole Smart Behaviour implementation packaging and runtime management within an OSGi bundle.

This could be done using DIANNE (DIANNE, s.f.). This is a modular software framework for designing, training and evaluating artificial neural networks that is already built on top of OSGi.

Another alternative would be to use Deeplearning4J (DL4J) (Deep Learning for Java, s.f.) which is now an Eclipse project. Currently it is not released as OSGi bundle, but this could be achieved within Brain-IoT.

The advantages of maintaining the same infrastructure that supports the dynamic deployment of software artefacts also for Smart Behaviours are clear. Not only the main goal would be achieved. It would be so in a clean way using the same design, APIs and packaging mechanism through the whole system.

But, in order to allow this option, a strong restriction must be observed: The Smart Behaviour implementations must be contained within the JVM or be managed and contained through JVM wrappers/containers.

6.4.4 Option: Python modules deployed within a Java / OSGi runtime

This could be approached using Jython (Jython: Python for the Java Platform, s.f.). An alternative could be JyNI (JyNI – Jython Native Interface, s.f.).

- The power of OSGi's Requirement/Capability model is available, as is the framework resolver.
- Python components are translated and run efficiently as native Java instructions.
- The communication between Python and Java components in OSGi framework would need some considerations.

However, as tempting as this option seems to be (allowing for an easy integration of Python ML stacks with JVM OSGi components), it has a very important drawback. Although there are many compatible Python compilers in different platforms, it is not feasible to use libraries like Numpy with any interpreter which is not the standard C based interpreter for Python (Cpython) or Cython (an optimising static compiler with support for language extensions for Python). This is because these libraries use optimized compiled algorithms and low-level operations written in the C extensions of the standard Python reference implementation.

Unfortunately, using Python for machine learning means using the Cpython compiler... But a viable alternative for this method has been suggested: (Jep - Java Embedded Python, s.f.) Jep embeds CPython in Java using JNI. Thanks to this, the limitation of Jython is already solved and support for Python modules like NumPy, Pandas, Tensorflow is provided.

Some limitations with some CPython extensions are noted (Workarounds for CPython Extensions, s.f.), and it must be explored how ML Python stack would suffer from this. But we consider Jep a viable option to integrate at least a subset of ML Smart Behaviours implemented in Python.

6.4.5 Considerations on the use of not only JVM technologies: the case of Python

Traditional consensus seems to be that Python AI algorithms are too slow to run in production. Also, in the context of BRAIN-IoT architecture, dependency management for Python libraries is a real issue: an initial deployment can be made to work but subsequent upgrades or changes face the same challenges.

However, in the ML field, especially in currently successful Deep Learning Neural Network libraries and frameworks, almost every one of them is based on an efficient C/C++ implementation (Lua in the case of Torch) that takes advantage of current GPU computational power if available.

Python façades and thin APIs are provided because:

- Python is a language with which data scientists and ML experts can be comfortable and proficient enough, abstracting developers and analysts from some of the low-level detail.
- It allows for direct integration with a C/C++ backend.

- It offers a great development environment to write prototypes and modify them while debugging, allowing for agile development cycle of ML models. This is essential in the field because, as stated in previous sections, trial and error of ML models is unavoidable and must be facilitated.

Python ML based technologies are both a strong example and a current real case of technology stacks that we think must be supported directly in BRAIN-IoT architecture. It would not align with this project goals and overall philosophy to restrict AI/ML developers and practitioners from using widely supported libraries and frameworks.

6.4.6 More considerations about the proposed options

If we were to use only Deep Learning Neural Networks, the use of DIANNE framework for OSGi/Java in production with a neutral neural network format from development/training could be the optimum approach.

But, to align to BRAIN-IoT goals and initial vision, and to support the broad set of techniques and implementations that Smart Behaviours should be able to use, we think the opaque artefact is a needed option, in order to be able to manage any possible implementation, despite de drawbacks of this approach.

Why not just use DIANNE along with one model interchange standard format (ONNX or NNEF) as main solution?

- DIANNE, best case scenario, only covers neural networks. We must always keep in mind that not all ML is based in neural networks. AI field is even broader. Furthermore, in many IoT real world scenarios NNs will not be a feasible solution due to legal issues, not being able to explain their outcomes.
- Some production environments that can be solved with ML models require a different lifecycle to: explore and train model -> export to ONNX / NNEF -> deploy to a DIANNE based artefact. For example: in some cases, model must be trained periodically in production environment. Bottom line is: Brain-IoT architecture should try to support any kind of development-production AI technology stack.
- ML field, particularly deep learning neural networks, is evolving very quickly right now. It seems a mistake to try to impose an evolving standard of NNs interoperability right now, as promising as it may seem…
- On the other hand, certainly both ONNX and NNEF are steps in the right direction and we do think they should be supported (but neither of them selected as the optimal interoperability format).
- While it is true that, given enough computational power (modern GPUs) and support for BLAS, CUDNN, etc., language choice is less important regarding efficiency and optimization[2] there are practical reasons behind the adoption of Python APIs and bindings for all these stacks. With the success these high-level APIs enjoy right now, it does not seem a good idea to not support those APIs in production environment, in a project that aims for interoperability with existing solutions.

Why not iPOPO or similar approach?

- If we gain little by comparison with the opaque artefact option, we think it is better to have both: the basic/generic method (allowing for a widely used practice, containerization, and integration with any available technology stack) and, optionally, a more ambitious one regarding architecture design (based on DIANNE probably).

[2] Also, all major deep learning frameworks and libraries are really implemented in C/C++ and use CUDA, BLAS, etc.

6.4.7 Conclusions and proposed solution

As a conclusion, these two options have been chosen and agreed upon:

1. An opaque artefact which is started in its own local container but managed by the BRAIN-IoT Edge Node JVM. Requiring the Edge Node to run a JVM and a Docker daemon.
2. A Java/OSGi assembly which runs within the Brain IoT Edge Node JVM, using DIANNE, DL4J, or Jep with Python modules. Requiring the Edge Node to run a JVM.

For Neural Networks:

- When possible, these are trained and released to production in one of the available neutral data formats (i.e. ONNX, NNEF). This will be a data only artefact and the runtime Edge Node must load any appropriate software or drivers to assemble the required ANN runtime considering local hardware capabilities (GPU, etc.). Using the option *2.- OSGi assembly*.
- For those cases where the trained ANN model cannot be export to a neutral data format, there are two options:
    a. The NN model is released with a non-neutral data format, specific to the NN technology stack being used, using the option *2.- OSGi assembly*.
    b. The NN model is released as part of the ANN component, using the option *1.- Opaque artefact*.

As this topic is now clarified, we are able now to begin developing simple runnable artefacts with Smart Behaviours in order to explore the required metadata for a precise and complete definition of each studied case. Some work has already been done in this regard, having agreed that this kind of Capabilities will be supported:

- Requirement or suggestion for GPU support
    - Optionally pointing specific vendor.
    - Optionally declaring a value for a given specific computing measurement metric.

Requirement of specific ML, linear algebra, etc. libraries (example: BLAS, cuDNN).

7 Conclusions

In this deliverable we defined Smart Behaviour and propose a list of abstractions for Artificial Intelligence elements that could and should be supported in BRAIN-IoT modelling tools, definition language, and runtime environment. These abstractions were rooted in a study of Artificial Intelligence field, considering current state-of-the-art Deep Learning Machine Learning techniques as a subset of that field, and distinguishing between classic algorithmic Artificial Intelligence, and Machine Learning methods where a model is trained in a supervised learning paradigm. Some existing well-designed libraries for Machine Learning were studied also, to point their main abstractions as inspiration and influence for our proposed design. A brief review of state-of-the-art libraries and frameworks was also presented.

An analysis of the currently described Use Cases for BRAIN-IoT scenarios (Water Critical Infrastructure and Service Robotics) was used to underline what kind of techniques could be used for each scenario where Artificial Intelligence is either needed or an optimal tool to solve the task.

Having discarded (at least for the first iterations) techniques for unsupervised learning like clustering, we try to generalize in our proposed design any possible method for:

- Algorithmic "classic" Symbolic Artificial Intelligence. Abstracted under the Intelligent Agent System paradigm where the agent processes inputs and returns actions and/or outputs.
- Machine Learning where human conducted feature extraction is needed and Deep Learning Machine Learning. Both abstracted under a general design where a model is trained with a Dataset and is the trained model processes new inputs to calculate a classification, regression or prediction.

With that initial design and set of abstractions, we propose also a small set of roles regarding how and when each abstracted element will be used in the Smart Behaviour development workflow. This can be used as basis for further work with the BRAIN-IoT modelling tools and IoT-ML development.

Several topics regarding runtime environment of Smart Behaviours and its relationship with the rest of BRAIN-IoT architecture are addressed as well. In these aspects we also kept the same goal and vision that motivates this project, which is: to allow for interoperability with not only currently successful technologies in the AI and ML field, but also to provide for future developments. Some conclusions and agreements were reached as to what are the viable and optimal options for: packaging, deployment, and "model" interchange format (when possible).

Appendix

A brief look at successful ML technologies

Deep Learning Neural Networks are having a tremendous impact right now, and therefore a chapter is used to briefly review existing successful Machine Learning frameworks and libraries, where most of them are DLNNs:

- Tensorflow. Framework for carrying out numerical computations using data flow graphs. Has an architecture that allows to distribute scalable computation on any CPU or GPU, be it desktop, server, and even mobile devices. Bindings for C++ and Python. Can be used through Keras. Has a great tool for visualization, Tensorboard. Supports distributed training.
- CNTK. Deep learning toolkit that describes neural networks as a series of computational steps via a directed graph, allowing parallelization across multiple servers and GPUs. Supports C++, C#, Java (experimental support still) and Python. Can be used through Keras. Supports distributed training.
- Caffe. Deep learning framework that works well on GPU for convolutional neural networks, but not as efficient for recurrent neural networks and other architectures. C++, binding for Python.
- Keras. High level neural network library written in Python. Not meant to be a full end-to-end framework but to serve as an interface and provide high level of abstraction. Supports CNTK and TensorFlow seamlessly and offers their advantages regarding the use of both CPU and GPU resources. Great for prototyping, probably should not be used for production (performance is always worse than using TensorFlow directly, and some flexibility is missed too by using Keras).
- Theano. No longer in development since September 2017.
- Torch. Library for scientific and numerical operations, great support of ML algorithms, powerful implementation of matrices operations, linear algebra routines and neural network models. Based on Lua, bindings for C and Python (PyTorch). Very flexible and efficient, makes use of GPU with very simple and clean API (DataParallel), using also a directed acyclic graph to define any model (but unlike Tensorflow, in Torch the graphs can be modified dynamically). This allows for a better debugging experience (again, a very important factor in ML models development), and offers additional possibilities for techniques like recursive neural networks. The API is cleaner and needs less boilerplate than Tensorflow. It is generally considered more developer friendly specially for debugging and it does not suffer from the penalization in performance (and flexibility) that the Tensorflow + Keras alternative does. Still lacks distributed training and a tool as powerful as the one Tensorboard offers for visualization. It has integration with Caffe2 through direct support of ONNX.
- Apache-MXNet. Deep learning framework, scalable to multiple GPUs and CPUs, supported by non-Google public cloud providers. Supports C++, Python, R, Scala, Perl and others. Support for ONNX models. Dynamic debugging (allowed in Torch for example), allows offline training and deployment of models to low-end devices with limited CPU and RAM.
- ScikitLearn. Python ML library, very useful for building and prototyping. Clean and elegant API, lots of algorithms, not the most efficient with GPU. Even if not used for ML models, it offers additional secondary but useful features that can complement other frameworks.
- Spark-MLlib. ML library usable in several languages (Java, Scala, Python, R). Interoperations with Numpy and R libraries. Can be plugged into Hadoop. Great option for large-scale data. Clean API inspired by ScikitLearn.
- MLPack. Scalable ML library fully implemented in C++ with binding for Python also. Simple API.
- Deeplearning4j. Deep learning library written for the JVM with implementations of many ML algorithms and APIs for Java, Scala, Clojure and even Python through Keras.
- OpenVINO. Toolkit based on convolutional neural networks that can import models from Tensorflow, MXNet or ONNX. Offers a common API to support execution across existing computer vision accelerators including Intel Movidious.

Some tools used by ML existing frameworks worth of mention:

- CUDA. NVIDIA language for programming GPUs, a feature that was and is a driving force of what is known as Deep Learning. Although CUDA it is not the only available option for GPU programming, it is the most popular and well supported right now across existing ML libraries and frameworks.
- BLAS. Specification for low level routines to perform linear algebra operations. Any high-level binding like Python Numpy library will use a BLAS library. There can be huge differences in performance between different BLAS libraries implementations.

Technologies for integration and configuration of production environments where ML models are to be used:

- Flask. Popular microservice framework used to expose on-demand ML models.
- Kubernetes. Orchestration system for containerized applications, very popular right now. Works with Docker and other container tools. Almost a must use to achieve scalability outside of public on-cloud solutions. It is also offered as a service in all major providers (MS, Google, Amazon).
- Spark. Framework for cluster computing, allowing data parallelism and fault tolerance across entire clusters. APIs for Scala, Java, Python and R. Modular architecture that includes MLlib, SQL API (this is quite recent) and Streaming to ingest data in mini-batches. Spark Dataframes are like the homonymous structures in other languages and libraries, but in Spark, these Dataframes can be distributed across a cluster. They are the common basic element of its API and thanks to that, the Spark APIs for the different supported languages offer the same performance.
- Kafka. A distributed streaming platform, which acts like a message broker, but storing streams of records in a fault-tolerant durable way. The communication runs over a simple TCP based protocol that has clients in many languages. Kafka is a platform suited to manage scalable and environments where heterogeneous clients can produce and consume data using the same model, in batch mode or real time, sharing the same model or updating the production environment model with new data sent to an analytics/training environment. It can be used to integrate different ML technologies also.
- ONNX. A definition for a computational graph model, operators and data types. Allows saving network weights and model definitions in a format supported by CNTK, PyTorch and Caffe, among others. Great direct support by all major non-Google ML actors. It also has converters for Tensorflow and Keras models (also: ScikitLearn, XGBoost, LibSVM).
- PMML. Definition for an interchange markup language to define predictive models produced by data mining and machine learning algorithms like SVMs, Naive Bayes, clustering, decision trees, regression models, and multi-layer neural networks. Trained models in TensorFlow can be exported in PMML format. It was a de facto standard for model interoperability but the rising of ONNX (pushed by Microsoft and Facebook) and its XML syntax are making it lose popularity and support among current libraries and frameworks. The same organization that defined PMML, the Data Mining Group, has created a successor, PFA, but it is unclear how much support it will gain.
- NNEF. A specification to enable interchange of the structure, operations and parameters of a trained neural network. Supported by most major hardware corporations (but not NVIDIA) among others. Its design seems to be more extensible than ONNX.

Acronyms

Acronym	Explanation
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
DLNN	Deep Learning Neural Network
IoT-ML	IoT Modelling Language
JVM	Java Virtual Machine
LLE	Local Linear Embedding
LSTM	Long Short-Term Memory
MAS	Multi Agent System
ML	Machine Learning
NN	Neural Network
NNEF	Neural Network Exchange Format
ONNX	Open Neural Network Exchange Format
OSGi	Open Services Gateway Initiative
PMML	Predictive Model Markup Language
PoC	Proof of Concept
SVM	Support Vector Machine
RNN	Recurrent Neural Network
UML	Unified Modelling Language
WoT	Web of Things

List of figures

Figure 1: Rule-based Symbolic AI systems, “classic” ML, Deep Learning 6
Figure 2: Example of an Intelligent Agent System design 8
Figure 3: Smart Behaviour overall system, first proposed design 19
Figure 4: Smart Behaviour design for a set of high-level abstractions 21

List of tables

Table 1: Smart Behaviour elements and its use for each proposed role 25

References

1. (s.f.). Obtenido de Jep - Java Embedded Python: https://github.com/ninia/jep
2. Brujin, & Lausen. (2005). *https://www.w3.org/Submission/WSML/.*
3. *CppMicroServices*. (s.f.). Obtenido de http://cppmicroservices.org
4. *Deep Learning for Java*. (s.f.). Obtenido de https://deeplearning4j.org/
5. *DIANNE*. (s.f.). Obtenido de http://dianne.intec.ugent.be/
6. *JADE*. (s.f.). Obtenido de http://jade.tilab.com/
7. *JyNI – Jython Native Interface*. (s.f.). Obtenido de https://jyni.org/#jyni-jython-native-interface
8. *Jython: Python for the Java Platform*. (s.f.). Obtenido de http://www.jython.org/
9. Kifer, Lausen, & Wu. (1995). Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the Association for Computing Machinery*.
10. McCarthy, Minsky, Shannon, & Rochester. (1955). *A PROPOSAL FOR THE DARTMOUTH SUMMER RESEARCH PROJECT ON ARTIFICIAL INTELLIGENCE.*
11. *NNEF*. (s.f.). Obtenido de https://www.khronos.org/nnef
12. Nwana. (1996). Software Agents: An Overview.
13. *ONNX*. (s.f.). Obtenido de https://onnx.ai/
14. OWL Working Group. (2012). *https://www.w3.org/OWL/.*
15. *PMML 4.3 - General Structure*. (s.f.). Obtenido de Data Mining Group: http://dmg.org/pmml/v4-3/GeneralStructure.html
16. *Prolog/Introduction to logic*. (s.f.). Obtenido de https://en.wikibooks.org/wiki/Prolog/Introduction_to_logic
17. Puget, J. F. (s.f.). *A Speed Comparison of C, Julia, Python, Numba and Cython on LU Factorization*. Obtenido de https://www.ibm.com/developerworks/community/blogs/jfp/entry/A_Comparison_Of_C_Julia_Python_Numba_Cython_Scipy_and_BLAS_on_LU_Factorization
18. Puget, J. F. (s.f.). *The Most Popular Language For Machine Learning Is …*. Obtenido de https://medium.com/inside-machine-learning/the-most-popular-language-for-machine-learning-is-46e2084e851b
19. RDF Working Group. (2014). *https://www.w3.org/RDF/.*
20. Russel, & Norvig. (s.f.). *Artificial Intelligence: A Modern Approach.*
21. *Scikit-Learn*. (s.f.). Obtenido de https://scikit-learn.org/stable/
22. Silver. (2017). Mastering the game of Go without human knowledge.
23. *Spark MLib*. (s.f.). Obtenido de https://spark.apache.org/mllib/
24. Tan. (2017). A brief history and technical review of the expert system research.
25. Voskoglou, C. (s.f.). *What is the best programming language for Machine Learning?* Obtenido de https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7
26. *What is iPOPO*. (s.f.). Obtenido de https://ipopo.readthedocs.io/en/0.8.1/foreword.html
27. *Workarounds for CPython Extensions*. (s.f.). Obtenido de Jep: https://github.com/ninia/jep/wiki/Workarounds-for-CPython-Extensions