Structural Monitoring with Distributed-Regional and Event-based NN-Decision Tree Learning using Mobile Multi-Agent Systems and common JavaScript platforms

Stefan Bosse

University of Bremen, Department of Mathematics & Computer Science,
ISIS Sensorial Materials Scientific Centre, Germany

Abstract Among the Internet-of-Things, one major field of application deploying agent-based sensor and information processing is Structural Load and Structural Health Monitoring (SLM/SHM) of mechanical structures. This work investigates a data processing approach for material-integrated and mobile ubiquitous SHM and SLM systems by using self-organizing mobile multi-agent systems (MAS), executed on a highly portable JavaScript-based Agent Processing Platform (APP), and optimized Machine Learning (ML) methods providing load class recognition from a set of sensors embedded in the technical structure. Machine learning approaches usually require a large amount of computational power and storage resources and ML is commonly performed off-line, not suitable for resource constrained sensor network implementations. Instead, a novel distributed-regional on-line learning is applied, with on-line distributed sensor processing and learning performed by the agent system. The APP provides ML as a service, and the agent itself only collects training and analysis data passed to the APP, finally returning a learned model that is saved by the agent in a compact format (and is available on any other location). A case study shows that the learning algorithm is suitable (stable) for noisy and time varying sensor data. Spatial global learning is reduced and mapped on local region learning with global voting.

Keywords: Structural Health Monitoring, Ubiquitous Perception, Sensor Network, Mobile Agent, Distributed Learning, Heterogeneous Networks, Embedded Systems

1. Introduction

One major field of application deploying agent-based sensor and information processing is Structural and Structural Health Monitoring (SM/SHM) of mechanical structures. Structural monitoring derive not just loads, but also their effects to the structure, its safety, and its functioning from sensor data. A load monitoring system (LM) can be considered as a sub-class of SHM, which provides spatial resolved information about loads (forces, moments, etc.) applied to a technical structure. The integration of SM systems in and the composition of SM systems with devices from the Internet-of-Things (IoT) is a breakthrough in future cloud-based sensor and information processing (see Fig. **1**), with mobile agents as an enabling technology.

One of the major challenges in SHM and LM is the derivation of meaningful information from sensor input. The sensor output of a SHM or LM system reflects the lowest level of information. Beside technical aspects of sensor integration the main issue in those applications is the derivation of a mapping function $F_m(S)$ which basically maps the raw sensor data input S, an n-dimensional vector consisting of n sensor values, to the desired information I, an m-dimensional result vector.

Basically there are two different information extraction approaches: (I.) First those methods based on a mechanical and numerical model of the technical structure, the device under test (DUT), and the sensor, and (II.) second those without any or with a partial physical model. The latter class can profit from artificial intelligence which usually bases on classification algorithms derived from supervised machine learning or pattern recognition using, for example, self-organizing systems like multi-agent systems with less or no a-priori knowledge of the environment.

Agents are already deployed successfully for scheduling tasks in production and manufacturing processes **[1]**, and newer trends poses the suitability of distributed agent-based systems for the control of manufacturing processes **[2]**, facing not only manufacturing, but maintenance, evolvable assembly systems, quality control, and energy management aspects, finally introducing the paradigm of industrial agents meeting the requirements of modern industrial applications by integrating sensor networks. Self-organization and adaptivity are central behaviours of complex distributed MAS. Mobile Multi-agent systems are already successfully deployed in sensing applications, e.g., structural load and health monitoring, with a partition in off- and online computations **[3]**. Distributed data mining and Map-Reduce algorithms are well suited for self-organizing MAS. Cloud-based computing with MAS, as a base for cloud-based design and manufacturing, means the virtualization of resources, i.e., storage, processing platforms, sensing data or generic information. Mobile Agents reflect a mobile service architecture.

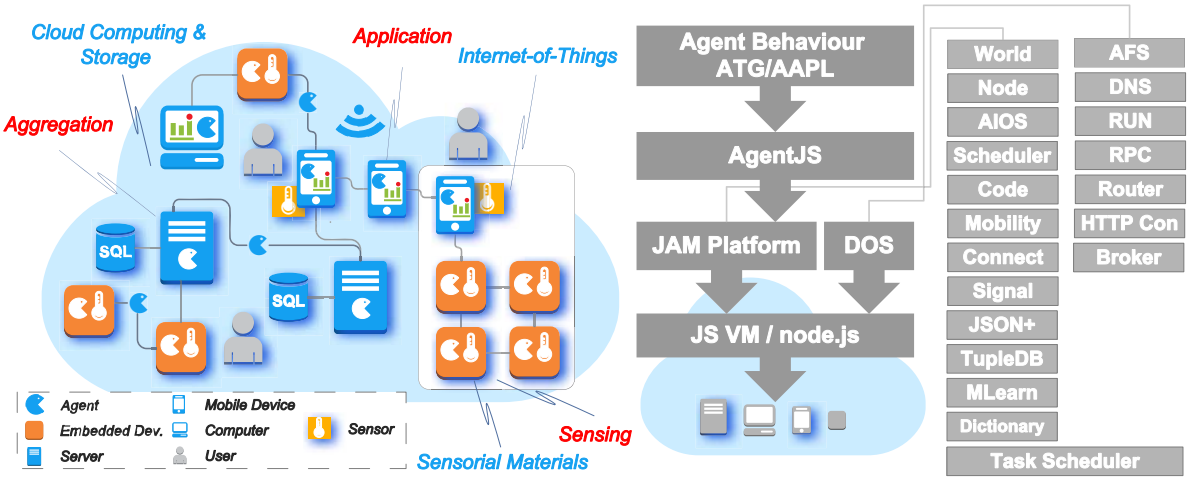

Fig. 1. (Left) Distributed Structural Monitoring, Perception, and Information processing, finally integrating SM networks in the IoT using Multi-agent Systems as a unified processing model. (Right) Components of the JavaScript Agent Machine (JAM), explained in Section 3..

The scalability of complex industrial applications and ubiquitous systems, e.g., networks of smart phones, using such large-scale cloud-based and wide area distributed networks deals with systems deploying thousands up to million agents. But the majority of current laboratory prototypes of MAS deal with less than 1000 agents **[2]**. Currently, many traditional processing platforms cannot yet handle a big number of agents with the robustness and efficiency required by the industry **[2]**. In the past decade the capabilities and the scalability of agent-based systems have increased substantially, especially addressing efficient processing of mobile agents. The integration of sensor networks in generic computer networks and the Internet raises communication and operational barriers which must be overcome by a unified agent processing architecture and framework, discussed in this work.

Multi-agent systems can be used for a decentralized and self-organizing approach of data processing in a distributed system like a sensor network, enabling information extraction, for example, based on pattern recognition, decomposing complex tasks in simpler cooperative agents

In this work the behaviour of mobile agents are modeled with Activity-Transition Graphs (ATG), which is directly implemented in JavaScript (*JS*) program code holding the entire control and data state of an agent (*AgentJS*), which can be modified by the agent itself using code morphing techniques (directly supported by *JavScript* JIT and VM platforms), and which is capable to migrate in the network between nodes. This approach requires only a minimal Agent Processing Platform Service (APPS) and a RPC-based Distributed Co-ordination Layer, entirely implemented in *JS*, too. The *AgentJS* code can be directly executed by the *JS* VM using the *JS* Agent Machine (*JAM*), in contrast to earlier work where special Agent *FORTH* code was used and executed on a dedicated stack-based VM (implemented itself in *JS*, but not limited to) **[4]**.

Agents processed on one particular node can interact and synchronize by using a tuple-space and a code dictionary. Remote interaction is provided by signals carrying data which can cross sensor node boundaries. The minimal APPS provides these interaction services among agent execution, mobility, agent role and privilege management, and Machine Learning services accessed over the *JAM* programming interface. This approach provides a high degree of computational independency from the underlying platform and other agents, and enhanced robustness of the entire heterogeneous environment in the presence of node, sensor, link, data processing, and communication failures.

Machine Learning with sensor data is always affected by noise, that can disturb the learning and classification process significantly, requiring suitable learning algorithms **[5][6]**. Furthermore, sensor data is usually available stream-based (even in the case of event-based sensor processing), demanding for incremental learning strategies interleaving learning (training) and analysis (prediction) phases. For example, Decision Tree Learning (DTL) is common algorithm used in sensor data processing. Furthermore, DTL is suitable for incremental learning, i.e., an incremental refinement of the learned model at run-time by back propagating the already learned model and new data, not requir-

ing to reuse the old sensor data sets already used in the old learned model **[7]**. Mobile agents are well suited for distributed learning, e.g., as shown in an adaptive monitoring sensor network in **[6]**.

The central approach in this work focuses on distributed machine learning for load classification by using mobile agents and the ability to support mobile reconfigurable code embedding the agent behaviour, the agent data, the agent configuration, and the current agent control state, finally encapsulated in portable *JavaScript* code. The mobility is granted by converting the agent program in a textual *JSON+* representation, and finally by parsing this text and executing the code again. This agent-specific mobile program code can be executed on a variety of different platforms including mobile devices, embedded devices, sensor nodes, and servers. The textual *JSON+* representation extends the *JS* Object Notation (*JSON*) with function code.

This work introduces some novelties compared to other data processing and agent platform approaches:

- Decentralized event-based sensor data processing and Machine Learning using mobile agents reduces computational and communication complexity and minimizes the overall network activity resulting in reduced energy consumption and increased robustness.
- Distributed-regional on-line learning and classification with pre-processed sensor data allows the prediction of the system response based on data from prior on-line training runs with selected load cases applied to the technical structure.
- Distributed global voting of regionally classified load situations with majority decision election and information diffusion behaviour increases robustness of the entire system.
- A novel hybrid Machine Learning algorithm suitable for noisy sensor data combines a modified Decision Tree Learning with ε-interval expansion, and nearest-neighbourhood look-up on prediction.
- A portable Agent Processing Platform with Machine Learning as a service and implemented entirely in JavaScript, suitable for a wide range of host platforms, e.g., embedded systems (sensor nodes), mobile devices (sensor nodes, smart phones), WEB browser, and servers.
- Agent mobility crossing different host platforms in strong heterogeneous networks, e.g., the Internet, and robust agent interaction by using tuple-space databases and global signal propagation, solving data distribution and synchronization issues in the design of large-scale distributed sensing and aggregation networks.

The next sections introduce the agent processing model, available mobility and interaction, and the proposed agent platform architecture related to the programming model. Finally, the sensor signal processing algorithms and the used novel learning methods are introduced and validated with use-case simulation results.

2. AgenJS: The Agent Behaviour and Interaction Model for Javascript

In this work, a novel agent process platform *JAM* is used that provides Machine Learning as a service for agents. *JAM* is implemented entirely in *JavaScript* (*JS*) including the ML service. The platform is explained in the next section. *JAM* is capable to execute agents programmed in *JS*, called *AgentJS*.

The behaviour of a reactive activity-based agent is characterized by an agent state, which is changed by activities. Activities perform perception, plan actions, and execute actions modifying the control and data state of the agent. Activities and transitions between activities are represented by an activity-transition graph (ATG)..

The agent behaviour and the action on the environment is encapsulated in agent classes, shown in Fig. **2**, with activities representing the control state of the agent reasoning engine, and conditional transitions connecting and enabling activities. Activities provide a procedural agent processing by a sequential execution of imperative data processing and control statements. Agents can be instantiated from a specific class at run-time. A multi-agent system composed of different agent classes enables the factorization of an overall global task in sub-tasks, with the objective of decomposing the resolution of a large problem into agents in which they communicate and cooperate with one other.

Agent interaction is required in MAS, providing synchronization and data exchange. This inter-agent communication should be on one hand abstract with respect to the underlying platform, network, and execution environment, but on the other hand it should be of practical and programmatical use. The tuple-space communication paradigm with a set of simple but synchronizing access operations (input ,output ,read, remove) is well accepted and an understood approach. Signals can be used instead for simple one-way notifications carrying no or simple data.

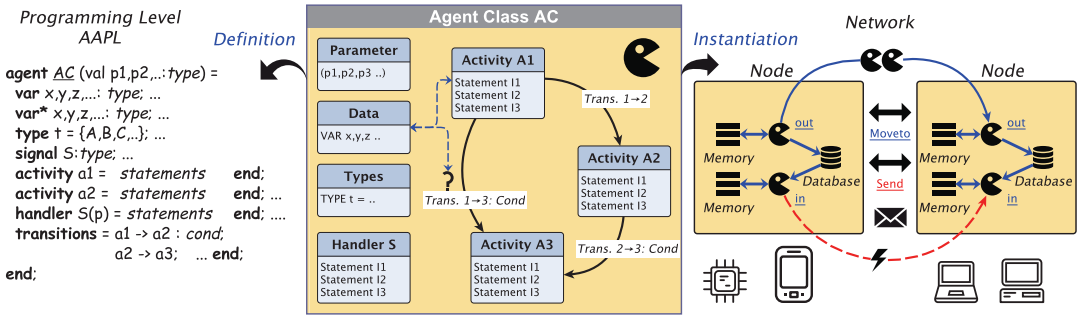

Fig. 2. Agent behaviour programming level with activities and transitions (AAPL, left); agent class model and activity-transition graphs (middle); agent instantiation, processing, and agent interaction on the network node level (right) **[8]**.

The activity-graph based agent model is attractive for fine-grained agent scheduling. An activity is always executed atomically, but after an activity terminates, it is a well defined break point for agent process scheduling.

An activity is activated by a transition depending on the evaluation of (private) agent data (conditional transition) related to a part of the agents belief in terms of the Belief-Desire-Intention (*BDI*) architecture, or using unconditional transitions (providing sequential composition), shown in Fig. **2**. Each agent belongs to a specific parameterizable agent class *AC*, specifying local agent data (only visible for the agent itself), types, signals, activities, signal handlers, and transitions. The principle *AgentJS* structure of an agent class is shown in Def. **1**.

In contrast to common *JS* objects, an *AgentJS* class definition may not use any references to free variables or functions. The `this` variable references always the agent object, and can be used, e.g., in transition functions, handlers, activities, and first order functions directly. *AgentJS* is a nearly syntactical and operational equivalent implementation of the ATG-based Agent Programming Language (*AAPL*, **[8]**) invented in earlier work.

```
1   var ac = function(p1,p2,..) {
2     this.p1=p1; .. Parameter
3     this.v1=0;  .. Varaiables
4     Activities
5     this.act = {
6       init: function () {..},
7       a1: function () {..},
8       a2: function () {..},
9       a3: function () {..},
10      a4: function () {..},
11      ..
12      end: function () {..}
13    };
14    Error and Signal Handler
15    this.on = {
16      error: function (e) {..},
17      exit: function (e) {..},
18      SIG1: function (v) {..},
19      ..
20    };
21    Transitions
22    this.trans = {
23      init: function () {return 'a1'},
24      a1: function () {..},
25      a2: function () {..},
26      ..
27    };
28    this.next='init';
29  }
```

Def. 1. Principle structure of an AgentJS Class Definition with a set of activities {a_1,a_2,..} encapsulated in an activity section, followed by the transition section implementing the agent ATG.

3. JAM: The JavaScript Agent Machine Processing Platform

JAM consists of different modules entirely implemented in *JS* that can be executed by any standalone *JS* VM or within WEB browsers, shown on the right side from Fig. **1**. The deployment in Internet and client-side applications like browsers and the Internet require a Distributed Co-ordination and Operation System layer (DOS) with a broker service, not discussed here (details can be found in **[4]**).

3.1. Agent Input/Output System AIOS

The *AIOS* is the main execution layer of *JAM*. It consists of the sandbox execution environment encapsulating an agent process, with different privileged sub-sets depending of an agent role (level 0,1,2). Furthermore, the *AIOS* module implements the agent process scheduler and provides the API for the logical (virtual) world and node composition. The sandbox environment provides restricted access to a code dictionary based on the privilege level, enabling code exchange between agents Level 0 agents are not privileged to replicate, create, or kill other agents and to modify their code.

3.2. Activity Blocking

In contrast to the *AAPL* model based on the Activity-Transition-Graph (ATG) model that supports multiple blocking statements (e.g., IO/tuple-space access) inside activities, *JS* is not capable of handling any kind of process blocking (there is no process and blocking concept). For this reason, scheduling blocks can be used in *AgentJS* activity functions handled by the *AIOS* scheduler. Blocking *AgentJS* functions returning a value use common callback functions to handle function results, e.g., `inp(pat,function(tup){..})`.

3.3. Agent Creation and the Sandbox Environment

Agents are either instantiated from an agent class template or forked from already existing agents. The template is genuine JS with some behavioural modifications, that can be transformed in the textual *JSON+* representation, derived from the *JS* Object Notification format (*JSON*), using a modified parser and text converter. *JSON+* includes additional function code. Agents are executed always in a sandbox environment, which requires always a code-text-code transformation that is performed on agent creation or migration, discussed below.

3.4. Agent Mobility

Agent mobility, provided by the *AIOS* `moveto(to)` statement, requires a process snapshot and the transfer of the data and control state of the agent process. The control state of an agent is stored in a reserved agent body variable `next`, pointing to the next activity to be executed. The data state of an *AgentJS* agent consists only of the body variables. Thus, the migration starts with a code-to-text transformation to the extended *JSON+* representation of the agent object, transportation of the text code to another logical or physical node, and a back text-to-code conversion with a new sandbox environment. The agent object is finally passed to the new node scheduler and can continue execution.

3.5. Agent Interaction

Agents can interact with each other by using a tuple-space database part of *JAM*. *AIOS* provides the common tuple-space access operations (`out(tup)`, storing a tuple *tup*, `inp(pat,function(tup){..})`, matching, returning, and removing a tuple based on pattern *pat*, `rd(pat,function(tup){..})`, matching and returning only, `rm(pat)`, removing a tuple only). Tuple space communication is generative, i.e., a tuple can survive the producer process/agent. For some situations, tuples can remain in the tuple space never consumed. To avoid a flooding of tuple spaces with "orphan" tuples, the `mark(tmo,tup)` operation can be used to store tuples with a limited lifetime `tmo`, which are destroyed by the TS manager automatically. These marking are extensively used in divide-and.-conquer systems discussed in the following sections. A signal `SIG` can be sent to an agent specified by its `ID` identifier using the `send(ID,SIG,arg)` statement. A signal can be send to a group of agents of a specified agent class `AC` and within a given local range Δ by using the `broadcast(AC,Δ,SIG,arg)` statement.

3.6. Machine Learning as a Service

Learning agents can access basic machine learning operations provided as a platform service, offered by `model= learn(datasets, classes, features, alg?)` and `feature = classify(model, dataset)` primitives.

The agent stores only the learned model, and do not carry any learning algorithms, leading to a separation of the learning algorithm (platform) from the data (agent).

4. SEJAM: The JavaScript Agent Simulator Environment

Commonly, execution and simulation platforms are completely different environments, and simulators are significantly slower in the agent execution compared to real-world agent processing on optimized processing platforms. *SEJAM* is a *JS-APPL* simulator implemented on top of the *JAM* platform layer, executing agents with the same VM as a standalone agent platform would do. This capability leads to a high-speed simulator, only slowed down by visualization tasks and user interaction. Furthermore, multiple simulators can be connected via a stream link (sockets, IP network connection, etc.), improving the simulation performance by supporting parallel agent processing. Furthermore, the simulator can be directly connected to any other *JAM* node.

The GUI of the simulator and the simulation world used in the case study section is shown in Fig. **3**. The GUI consists of the simulation world, composed of 64 logical nodes connected with virtual circuit links. Each node shape provides information about the node name in the first row, the number of agents and tuples in the database in the second row, and some flag indicators in the last row, e.g., flags signalling the existence of specific agents or sensor values.

On the right side there is a code and data navigator. Each node can be selected including the world object. The code navigator can be used to explore node and agent information in *JSON* tree presentation. The bottom part of the simulator contains a logging and message window. Agents can write messages to this window, and a compacted *JSON* can be printed from selected items in the object navigator tree. Furthermore, agents executed in the simulator world inherit a special simulation object, which can be used to get specific simulation and world information, e.g., the current simulation step, or support for creation of agents on a specific node, e.g., used by the world agent, that is the only agent created and started at the beginning of the simulation. Multiple simulator worlds (*SEJAM* instances) can be connected enabling the composition of complex simulation worlds.

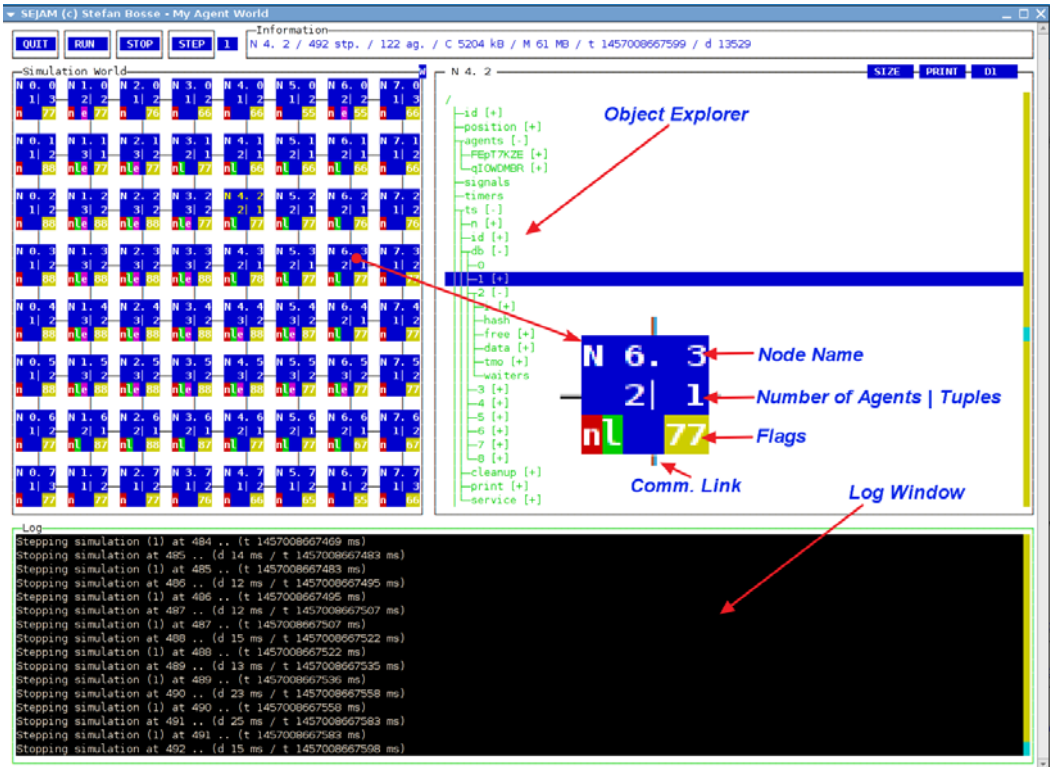

Fig. 3. SEJAM simulation demonstration with a simulation world consisting of 8x8 logical nodes populated with mobile and non-mobile agents, indicated by markings on the bottom of the node shape (blue rectangle).

5. Distributed Event-based Sensing and Learning with Multi-Agent Systems

Large scale sensor networks with hundreds and thousands of sensor nodes require data processing concepts far beyond the traditional centralized approach with request-reply interaction. Decentralized mobile Multi-Agent systems can be used to implement smart and optimized sensor data processing in these distributed sensor networks.

5.1. Event-based versa streamed Sensor Processing

There are still many sensing applications operating stream-based, i.e., the sensor information is collected by one or multiple dedicated nodes periodically from all sensor nodes, requiring high-bandwidth communication and consuming a significant amount of power. Frequently, most of the sampled sensor data do not contribute to new information about the sensing system, in a multi-sensor system only a few sensors will change their data beyond a noise margin. For example, there is no change in the load of a mechanical structure, and hence the is no significant change in the sensor data set. Or a change of the load situation results in a sensor data change in a spatially limited region, not effecting other regions.

In previous work **[3]** it was shown that three different data processing and distribution approaches can be used and implemented with agents, leading to a significant decrease in network communication activity and a significant increase of the reliability and Quality-of-Service.

1. An event-based sensor distribution behaviour is used to deliver sensor information from source sensor to computation nodes based on local decision and sensor change predication.
2. Adaptive path finding (routing) supports agent migration in unreliable networks with missing links or nodes by using a hybrid approach of random and attractive walk behaviour
3. Self-organizing agent systems with exploration, distribution, replication, and interval voting behaviours based on feature marking are used to identify a region of interest (ROI, a collection of stimulated sensors) and to distinguish sensor failures (noise) from correlated sensor activity within this ROI.

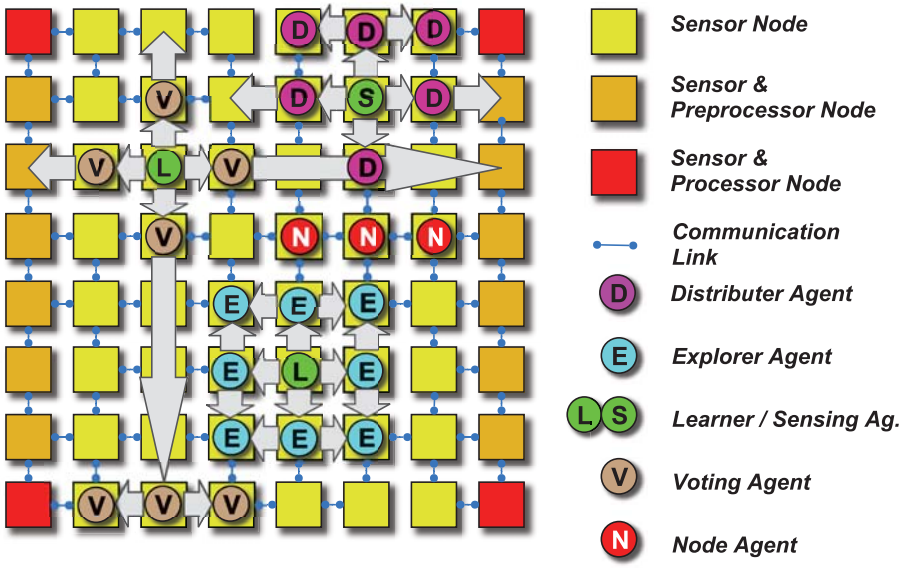

Fig. 4. The logical view of a sensor network with a two-dimensional mesh-grid topology (left) and examples of the population with different mobile and immobile agents (right): node, learner, explorer, and voting agents. The sensor network can contain missing or broken links between neighbour nodes. Non-mobile node agents are present on each node. Pure sensor nodes (yellow nodes in the inner square) create learner agents performing regional learning and classification. Each sensor node has a set of sensors attached to the node, e.g., two orthogonal placed strain gauge sensors measuring the strain of a mechanical structure..

In Structural Monitoring applications, sensor nodes are commonly arranged in some kind of a two-dimensional grid network (as shown in Fig. **4**) and they provide spatially resolved and distributed sensing information of the surrounding technical structure, for example, a metal plate or a composite material. Usually a single sensor cannot provide any meaningful information of the mechanical structure .In contrast to previous work that used inverse numerical computation to determine load situations, in this work instead spatially bounded regions in the network, Regions of Interest (ROI), are used to compute event-based a prediction and classification of the load case situation using supervised machine learning. Again, mobile agents are used to collect (percept) and deliver sensor data, but only limited to the ROI, shown in Fig. **4** (explorer agents delivering neighbourhood sensor data to learner agents).

Fig. **5** gives an overview of the composition of the complete sensor processing and distributed learning system with different agent classes, discussed in the following sections. Some classes are super classes composed of sub-classes (e.g. the learner and the explorer class). A sensor node is managed by a non-mobile node agent, which creates and manages a sampling and sensing agent, responsible for local sensor processing, and a learner agent, which is initially inactive. The world class is only used in the simulation environment and has the purpose to create and initialize the sensor network world and to control the simulation using monte-carlo techniques. The notify (`todo`) agents are injected in the network to notify nodes and learner agents about the network mode, if it is in training mode and which training class (load situation) is currently applied, or being in the classification mode. The notify agents will replicate and diffuse in the network (divide and conquer behaviour).

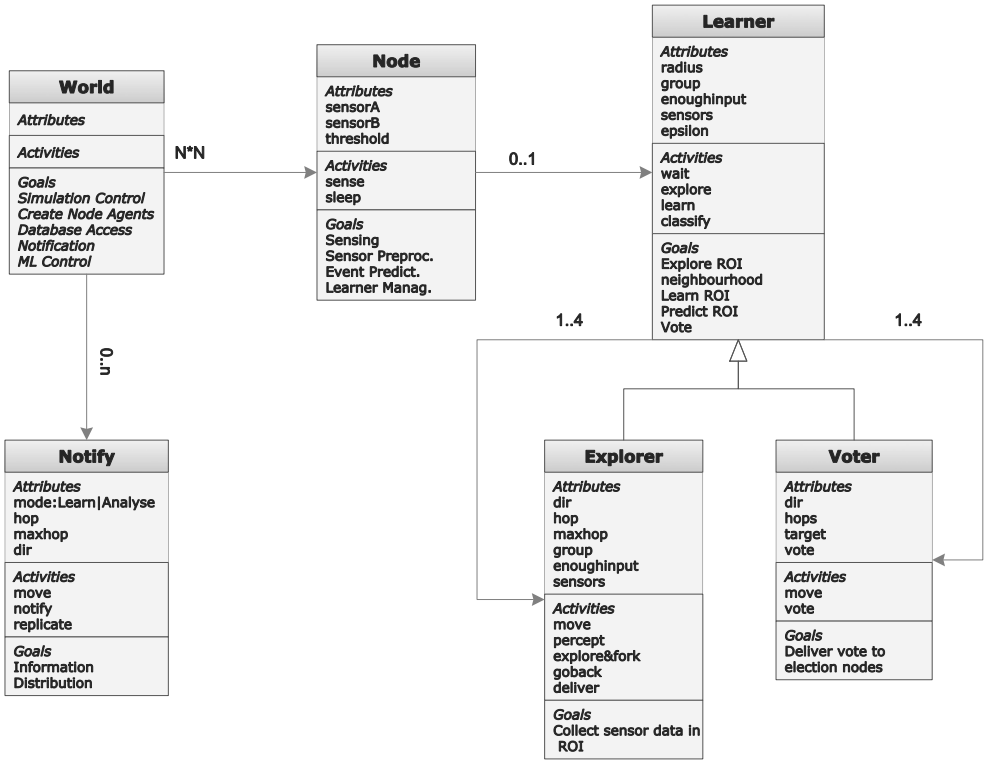

Fig. 5. Overview of different agent classes and sub-classes used for the sensor processing and learning in the network and their relationships (gray solid arrow: agent instantiation at run-time, light arrow: sub-class relationship). The world agent is only used in the simulation and handles the physical world and the network.

5.2. Regional-distributed Event-based Learning

Distributed learning divides a spatial distributed data set in local regions and applies learning to the limited local regions, based on a divide and conquer approach. Decision trees are simple models derived from learning with training data, and well suited for agent-based learning. A learned model is used to map data set vectors on class values. The tree consists of nodes testing a specific attribute variable, i.e., a particular sensor value, creating a path to the

leaves of the tree containing the classification result, e.g., the load situation class. Among the distribution of the entire learning problem, event-based activation of learning entities can improve the system efficiency significantly. Commonly the locally sampled sensor values are used for an event prediction.

The event-based regional learning leads to a set of classification results, which can differ significantly, i.e., the classification set can contain wrong predictions. To filter out and suppress these wrong predictions, a global major vote election is applied. All nodes performed a regional classification send their result to the network collecting all votes and perform an election. This election result is finally used for the load case prediction. The variance of different votes can be an indicator for the trust of the election giving the right prediction.

5.3. ε-Entropy-σ NN-Decision Tree Learning

Traditional Decision Tree Learner (DTL) (e.g., using the *C4.5* algorithm) select data set attributes (feature variables) for decision making only based on information-theoretic entropy calculation to determine the impurity of training set columns (i.e., the gain), which is well suited for non-metric symbolic attribute values, like color names, shapes, and so on. The distinction probability of two different symbols is usually 1. Numerical sensor data is noisy and underlies variations due to the measuring process and the physical world. Two numeric (sensor) values *a* and *b* have only a high distinction probability if the uncertainty intervals [*a*-σ,*a*+σ] and [*b*-σ,*b*+σ] due not overlap. That means, not only the entropy of a data set column is relevant for numerical data, the standard deviation σ and value spreading of a specific column must be considered, too. To improve attribute selection for optimized data set splitting, a column ε-interval entropy computation was introduced, that extends each value of a column vector with an uncertainty interval $[v_i\text{-}\varepsilon, v_i\text{+}\varepsilon]$. Values with overlapping intervals are considered to be non distinguishable, lowering the entropy *entropy* (with $\lfloor$x: lower bound of a value/interval, $\lceil$x: upper bound, and |v| as the size of a vector), with the computation given by Eq. **1**.

$$
\begin{aligned}
\mathrm{entropy}_\varepsilon(cols,\varepsilon) &= \sum_{i=1..|cols|} -\mathrm{prob}_\varepsilon(col_i,cols,\varepsilon)\log_2(\mathrm{prob}_\varepsilon(col_i,cols,\varepsilon)) \\
\mathrm{prob}_\varepsilon(v,cols,\varepsilon) &= \frac{\sum_{i=1..|cols|}\begin{cases}0:\mathrm{overlap}([cols_i-\varepsilon,cols_i+\varepsilon],[v-\varepsilon,v+\varepsilon]) \\ 1:otherwise\end{cases}}{|cols|} \\
\mathrm{overlap}(v_1,v_2) &= \begin{cases}true:(\lceil v_1 \geq \lfloor v_2 \wedge \lceil v_1 \leq \lceil v_2\ ) \vee \\ \qquad (\lceil v_2 \geq \lfloor v_1 \wedge \lceil v_2 \leq \lceil v_1\ ) \\ false:otherwise\end{cases} \\
\mathrm{distance}(v_1,v_2) &= \left|\frac{\lfloor v_1+\lceil v_1}{2}-\frac{\lfloor v_2+\lceil v_2}{2}\right|
\end{aligned}
\tag{1}
$$

The ε-entropy is calculated for all data set columns, and the attribute (feature variable) for the column with highest entropy value is selected. The column can still contain non-distinguishable values with overlapping 2ε intervals. All overlapping 2ε values are grouped in partitions that cannot be classified (separated) by the currently selected attribute variable. Only partitions - ideally containing only one data set value - are used for a classification selection. All data sets in one partition create a sub-tree of the current decision tree node. If there is only one partition available (containing more than one class target, a data set attribute selection is based on the column with the highest standard deviation, but the 2ε separation cannot be guaranteed in this case, lowering the prediction accuracy. The basic principle of the learning algorithm, which is an adaptation of a common discrete C4.5 Decision Tree Learner, is shown in Alg **1**. The extended algorithms can be found in **Appendix A.**. It creates a model based on attribute value interval selection, e.g. x∈[500..540], instead the commonly used and simplified relational value selection, e.g., x < 540, which is an inadmissible extrapolation beyond the training set boundaries and prevent recognizing totally non-matching data.

Alg. 1. Principle Learning and Classification algorithms. The entropy computation applying the 2ε interval to values is shown in Eq. **1**..

```
type value = number | number range
The learned model is a decision tree with nodes and leaves
type model = Result (name: string) |
             Feature (name:string, featvals: model array) |
             Feature Value(val: value, child: model)

function createTree(datasets, target, features)
  1. Select all columns in the data set array with the target key
  2. If there is only one column, return a result leaf node with the target
  3. Determine the best features by applying entropy and value deviation computation
  4. Select the best feature by maximal entropy
  5. Create partitions from all possible column values for this feature
  6. If there is only one partition holding all values, go to step 10
  7. For each partition create a child feature value node
  8. For each child node apply the createTree function with the
     remaining reduced data set by filtering all data rows containing at least
     one value of the partition in the respective feature column of the data set,
     and by using a reduced remaining feature set w/o the current feature
  9. Return a feature node with previously created feature value child nodes. Finished.

  10. Select the best feature by maximal value deviation
  11. Merge overlapping or equal column values
  12. For each possible value create a feature value node
  13. For each child node apply the createTree function with the
      remaining reduced data set by filtering all data rows containing at least
      one value of the partition in the respective feature column of the data set,
      and by using a reduced remaining feature set w/o the current feature
  14. Return a feature node with previously created feature value child nodes. Finished.
end

function classify (model,dataset)
  I. Iterate the model tree until a result leaf is found.
  II. Evaluate a feature node by finding the best matching feature value node for the
      current feature attribute by finding the feature value with minimal distance
      to the current sample value from the data set.
end
```

The prediction (analysis and classification) algorithm is a hybrid approach, too. It consists of the tree iteration, but uses a simple nearest-neighbourhood estimation for selecting the best matching feature value with a given sample sensor value.

The learned DT is composed of result leaves and feature/feature value selection nodes. A learned DT model can be easily transformed in a table representation, enabling the implementation of learned DT models on hardware level using simple linear Look-up Tables (LUT), illustrated in Fig. **6**.

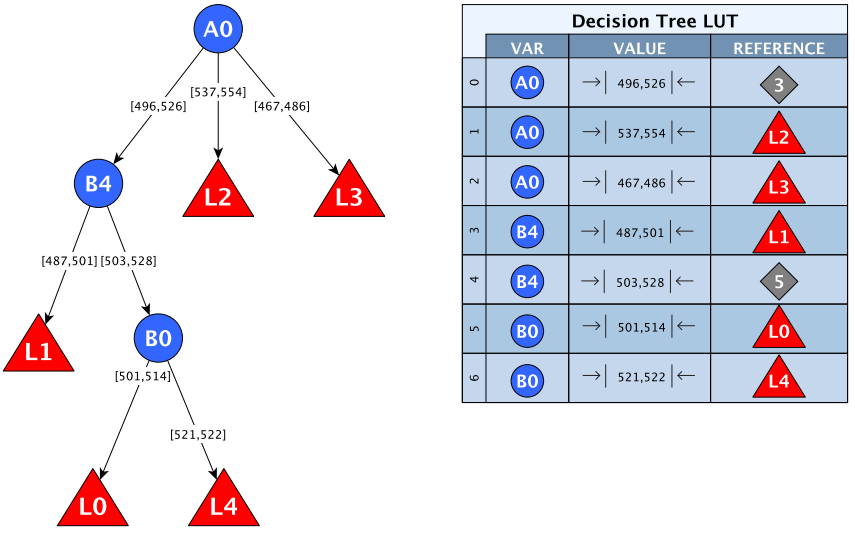

Fig. 6. A learned decision tree and its linear look-up table representation (A_i: Sensor A on node i, B_i: Sensor B on node i, L_i: Load case i)

6. A Case Study: Structural Load Monitoring

The distributed and event-based machine learning introduced in the previous section is evaluated with load data from previous work **[3]**. In this work, different load situations were applied to a metal plate, and the strain of the plate was computed at particular points using FEM simulation, finally mapped on artificial sensor data processed by a MAS in the *SEJAM* MAS simulator introduced in Sec. 4. The structure of the sensor network was already shown in Fig. **4**.

A load-strain matrix $\boldsymbol{T}$ must be computed to compute a strain matrix from a specific load situation (details can be found in **[3]**). To compute $\boldsymbol{T}$, the simulated loads are cylindrical weights placed at N=400 weight positions from an equidistant rectangular grid for $N_x = N_y$=20. The force on the upper surface of the upper horizontal plate due to the loading hence vanishes outside the circle covered by the weight; inside this circle the force points in direction $-\boldsymbol{z}$ and equals 1 N/cm^2. Since the used deformation model is linear, the actual value assigned to this force is unimportant if it re-scale the load-strain matrix such that the magnitude of reconstructed loads matches those of the true load for noise-free data. After computing the deformation field, the surface strain in $\boldsymbol{x}$ and $\boldsymbol{y}$ direction at the sensor points was extracted by computing the deformation field and the extracted surface strain for a sequence of refined meshes.

Apart from the simple cylindrical loads to compute the load-strain matrix $\boldsymbol{T}$, five pairwise different loads $\boldsymbol{l}^{(1)}$, .. , $\boldsymbol{l}^{(5)}$ with different characteristics and acting on different parts of the steel plate were simulated, shown in Fig. **7**.

The simulated strain values are then converted into integer values in between 1 and 1024, with a zero signal corresponding to the value 512; if σ_i denotes a simulated strain value, this conversion is done using the formula $\boldsymbol{s_i} = \lfloor 512 + 10000*\sigma_i \rfloor$, $1 \leq i \leq 2M$. ($\lfloor a \rfloor$ denotes the largest integer smaller than or equal to $a \in \mathbb{R}$.).

Thus, five strain measurement vectors $\boldsymbol{s}^{(1)}$, ... , $\boldsymbol{s}^{(5)}$ are computed., and these five data sets were feed consecutively as sensor values into the simulation framework for the sensor network shown in Fig. **4**.

After a randomly chosen load situation was applied, the response of the LM is evaluated. Between two load cases there is always the null-load case that is applied to relax the system. For the learning an ε=5 setting was used. Montecarlo simulation of sensor noise was applied with ε=5, too, adding equally distributed noise intervals [-ε,ε] to the sensor values.

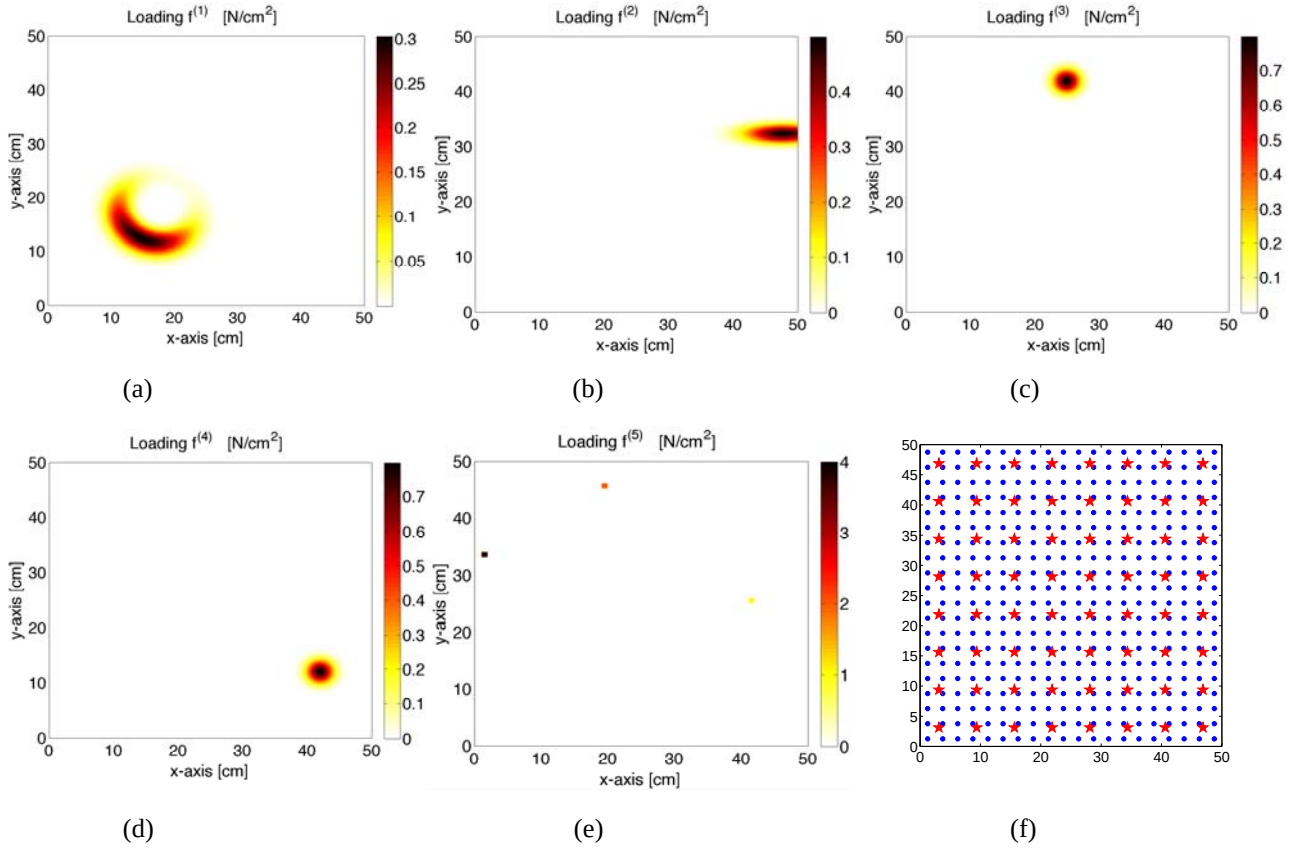

Fig. 7. From (a) to (e): The five loads $l^{(1)}, .. , l^{(5)}$. (f) The positions of the 400 weight points $w\{i,j\}$ are indicated using blue dots; red crosses indicate the positions of the 64 sensors **[3]**.

Simulation results are shown in Fig. **8**. The top figure shows the temporal agent population for a long-time run with a large set of single training and classification runs, with a zoom shown in the middle two figures. Each peak represents a particular training or classification run. The learner agents are non-mobile, and hence the population do not change in time. Side and edge nodes are not populated with learner agents. A learner agent covers the ROI containing its host node and eight surrounding nodes. If the host node detects a sensor event (change), it notifies the waiting learner agents by storing a `TODO` tuple in the database, consumed by the learner agent. Either a training (learning) or classification (prediction) request is send. In both operational cases the learner agent will send out explorer agents to collect sensor data in the neighbourhood, which is back delivered to the learner. In the classification modus the learner will use the already trained and learned model to predict the load case situation. The result is send out in the network by using voting agents, finally accumulated by the four edge nodes of the network by the major vote election.

The bottom figure shows global classification results obtained by major voting of all event-activated regional learner agents. The load sequence was randomly chosen, but always with a idle load situation (l^0) inserted between two different load cases. The learner must predict this load after a change in the load situation, too, meaning there is no load applied to the structure. The major election results show a very accurate prediction of significant distinguishable loads, i.e., l^1, l^2, l^3, and l^4. The last load case l^5 is hard to distinguish from the zero load case due to the weak point forces relative to the noise level.

Each learning/classification run requires about 0.5-1MB communication costs (using code compression) in the entire network only, and the agent population reaches up to 400 agents (peak value, but executed in the simulation by one physical *JAM* node), and a logical *JAM* node is populated with up to 10 agents.

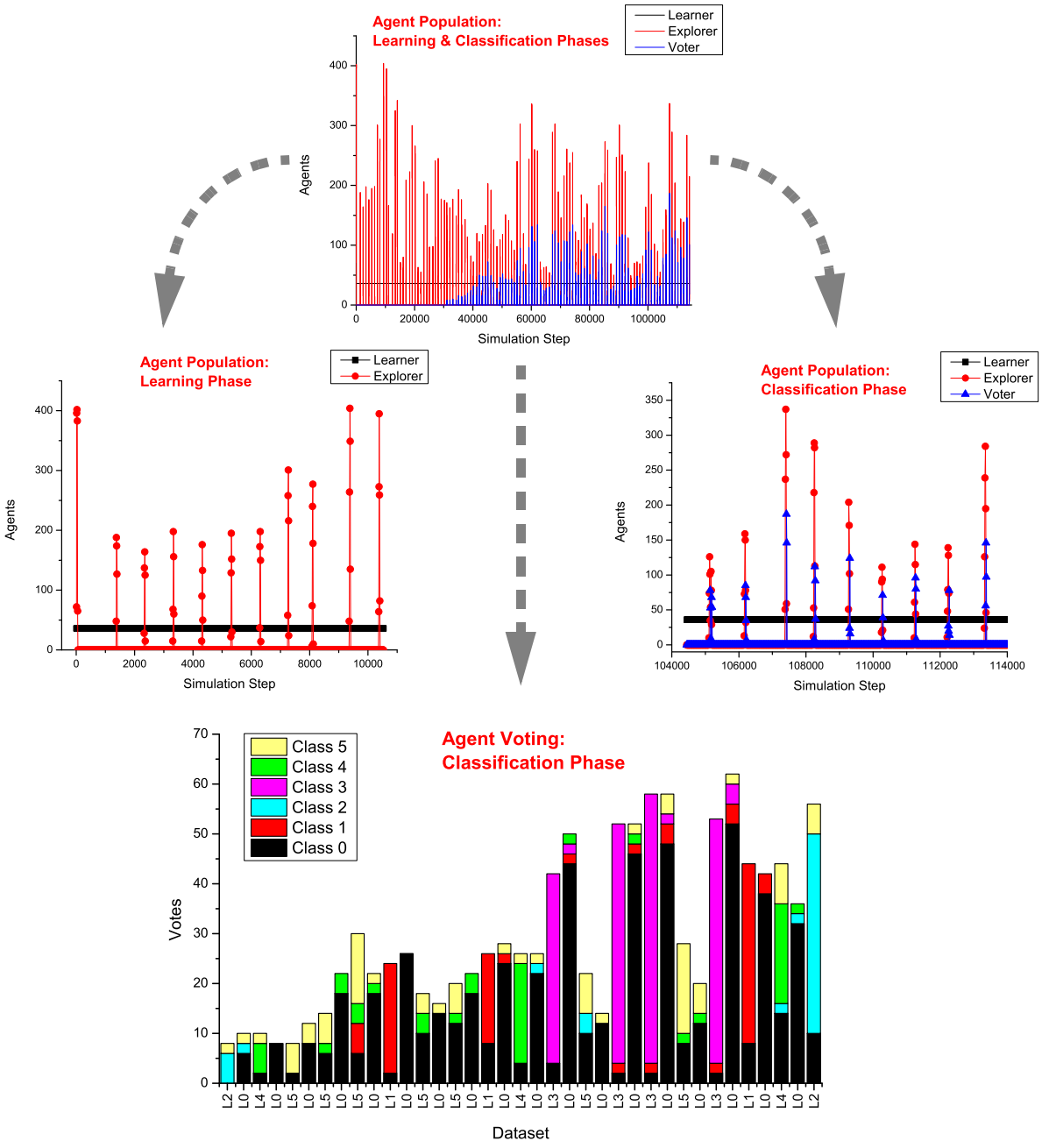

Fig. 8. Simulation Results. The top figure shows the temporal agent population for a long-time run with a large set of single training and classification runs, with a zoom shown in the middle two figures. The bottom figure shows global classification results obtained by major voting of all event-activated regional learner agents.

7. Conclusions and Outlook

A novel hybrid and distributed Machine Learning approach for noisy sensor data was presented and evaluated. The ML approach utilizes a modified Decision Tree learner with feature selection based on a 2ε interval entropy and deviation computation. The learned model is compact and contains feature value ranges instead of single values. It separates variable values at least by 2ε intervals giving enhanced noise immunity and classification stability. At a specific feature variable, non distinguishable class targets are partitioned and classified with deeper tree branches.

The classifier selects the best matching feature value by finding the nearest neighbour of a set of feature values for a specific feature variable. Agents are used for learner in a spatial limited region (ROI). This local classification is event-based triggered based on local sensor prediction. Finally, a group of learner agents send a voting agent to dedicated election nodes, making a major decision for the load class situation. The simulation experiments with simulated load situation sensor data shows accurate prediction results, even in the case of noise. A novel pure *JavaScript* Agent Machine (*JAM*) capable of executing mobile agents entirely programmed in *JavaScript* was introduced and used in the simulation. The ML is a service provided by the platform accessed by the agents using the platform API. Hence, the agents store only the learned model, and not the algorithms.

Though the benefits of incremental learning was outlined in the introduction, some more work must be investigated to transform the proposed DTL to an incremental version. Currently, the agents must store the entire training data set history and perform the model learning each time with the full set of training data.

8. References

[1] M. Caridi and A. Sianesi, *Multi-agent systems in production planning and control: An application to the scheduling of mixed-model assembly lines*, Int. J. Production Economics, vol. 68, pp. 29–42, 2000.

[2] M. Pechoucek, V. Marík, 2008. *Industrial deployment of multi-agent technologies: review and selected case studies.* Auton. Agent. Multi-Agent Syst. 17 (3), 397–431

[3] S. Bosse, A. Lechleiter, *A hybrid approach for Structural Monitoring with self-organizing multi-agent systems and inverse numerical methods in material-embedded sensor networks*, Mechatronics, 2015, doi:10.1016/j.mechatronics.2015.08.005, in press

[4] S. Bosse, *Unified Distributed Computing and Co-ordination in Pervasive/Ubiquitous Networks with Mobile Multi-Agent Systems using a Modular and Portable Agent Code Processing Platform*, in The 6th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2015), Procedia Computer Science, 2015.

[5] A. Moraru, M. Pesko, M. Porcius, D. Mladenic, and C. Fortuna1, *Using Machine Learning on Sensor Data*, Journal of Computing and Information Technology, vol. 4, pp. 341–347, 2010.

[6] W. Liu, *DATA DRIVEN INTELLIGENT AGENT NETWORKS FOR ADAPTIVE MONITORING AND CONTROL*, Michigan Technological University, 2012.

[7] J. Gama and P. Kosina, *Learning Decision Rules from Data Streams*, Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, 2011.

[8] S. Bosse, *Design and Simulation of a Low- Resource Processing Platform for Mobile Multi-Agent Systems in Distributed Heterogeneous Networks*, Agents and Artificial Intelligence, LNAI 8946, Springer, Béatrice Duval, J. van den Herik, S. Loiseau, and J. Filipe, Eds. Springer, 2015.

Appendix A.

Alg. 2. Learning Algorithm creating a decision tree. The matrix S contains sensor values delivered from a specific ROI, and there are two sensors a/b on each node available. Notation: |v| is the size of a vector or object, ⌊:lower bound, ⌈: upper bound of an interval, ⌊v⌋: range (min/max) of a list/vector, {}: list or set or labeled object, []: list vector, `entropy` and `overlap` are defined in Eq. **1.**, the first element of an array/list has index 1, | is a condition, : A → B applies a mapping function, v_i: i-th element of v

```
1   A value can be a number or a range, processed by the following functions
2   type value = number | [number,number]
3   The learned model is a decision tree with nodes and leaves
4   type model = Result(name:string) |
5                Feature(name:string,featvals: model []) |
6                Feature_value(val:value,child:model)
7
8   type class = string
9
10  function collect(radius:number): data {}
11    N := 2*radius
12    NN := N*N
13    Collect all sensor data from nodes within radius hops and store them in a matrix
14    S := N x N vector matrix, delivered and collected from explorer agents;
15    Create a labeled feature object; there are two sensors a/b on each node available
16    return
17     {a1:S[-radius,radius]['a'],   b1:S[-radius,radius]['b'],
18      a2:S[-radius+1,radius]['a'], b2:S[-radius+1,radius]['b'], ..
19      aNN:..,bNN:..}
20  end
21
```

```
22  Recursively create the decision tree
23  function createTree(data:{}[], target:string, features:string []): model
24
25    Find the best feature in the current dataset from the given feature list
26    function getBestFeatures(data: {}[],class:string,features:string [],ε): {}[]
27      function deviation(vals:value [])
28        mu := (∑ {val vals: (⌊val + ⌈val)/2})/|vals|
29        dev := (∑ {val vals: ((⌊val + ⌈val)/2-mu)})^2)
30        return dev
31      end
32      bestfeatures=[];
33      ∀ feature ∈ features do
34        Get the column vector from the dataset with the given feature
35        col := select feature from data
36        e := entropy(col,ε)
37        d := deviation(col)
38        add {entropy:e,deviation:d,range:⌊col⌋,name:feature} to bestfeatures
39      done
40      Sort beastfueatures with decreasing entropy
41      return { f_i,f_j ∈ bestfeatures | f_i.e > f_j.e ∧ i<j }
42    end
43
44    Create a partition list with groups of values.
45    Each parition must be separated at least 2ε
46    function partitionVals(vals: value [],ε): value [][]
47      partitions := []; partition := []
48      ∀ i ∈ {1 .. |vals|-1} do
49        Take two neighbour values, values are ordered, first index is 1, increasing i
50        if ⌈vals_i < ⌈(vals_i+1 + 2ε) then add vals_i to partition
51        else
52          add partition to partitions
53          partition := [vals_i]
54        end
55      done
56      add vals_i to partition; add partition to partitions
57      return partitions
58    end
59
60    function getPossibleVals(data:{}[],feature:string) : []
61      Get the column vector from the dataset with the given feature
62      col := select feature from data
63      Sort column with decreasing values
64      return { v_i,v_j ∈ col | ⌈v_i > ⌊v_j ∧ i<j }
65    end
66
67    targets := select target from data;
68    if |targets| = 1 then return Result(targets_1)
69    bestFeatures := getBestFeatures(data, target, features, ε)
70    bestFeature := bestFeatures_1
71    Create remaining feature name list
72    remainingFeatures := { f ∈ bestFeatures | f.name ≠ bestFeature.name : f → f.name }
73    possibleValues := getPossibleVals(data,bestFeature.name)
74
75    treevalues := []
76    partitions := partitionVals(possibleValues,ε)
77    if |partitions| = 1 then
78      no further 2ε separation possible, find best feature by largest distance,
79      resort best feature list with respect to the value deviation
80      bestFeatures := { f_i,f_j ∈ bestFeatures | f_i.d > f_j.d ∧ i<j }
81      bestFeature := bestFeatures_1
82      remainingFeatures := { f ∈ bestFeatures | f.name ≠ bestFeature.name : f → f.name }
83      possibleValues := mergeVals(getPossibleVals(data,bestFeature.name))
84      ∀ val ∈ possibleValues do
85        Filter all data rows containing the current value in feature column
86        reduceddata := { row ∈ data | overlap(val,row[bestFeature.name] }
87        child_node := Value(val,createTree(reduceddata, target, remainingFeatures,ε));
88        add child_node to treevalues
89      done
90    else
91      ∀ part ∈ partitions do
```

```
92        Filter all data rows containing at least one value in the part. in feature column
93        reduceddata := { row ∈ data | ∃ v ∈ part • overlap(v,row[bestFeature.name] }
94        child_node := Value([⌊partition-ε, ⌈partition+ε],
95                            createTree(reduceddata, target, remainingFeatures,ε))
96        add child_node to treevalues
97      done
98    end
99    return Feature(bestFeature.name,treevalues)
100 end
101
102 function learn(training_data:{}[], targetkey:string, features:string []): model
103   return createTree(training_data,targetkey,features)
104 end
```

Alg. 3. Classification (prediction) Algorithm using the previously learned classification model.

```
1
2   function classifiy(model,dataset): class
3
4     function nearestVal(vals: model [],sample): model
5       best := none
6       ∀ val ∈ vals do
7         d := distance(val.val,sample);
8         if best = none ∨ best.d > d then best := {val:val,d:d} end
9       done
10      if best ≠ none then return best.val else return none;
11    end
12    root := model
13    while root ≠ none ∧ root ≠ Result do
14      attr := root.name
15      sampleVal := select attr from dataset
16      childNode := nearestVal(root.vals,sampleVal)
17      if childNode ≠ none then root := childNode.child
18      else root := none end
19    done
20    if root ≠ none then return root.name else return none end
21  end
22
```