Chapter 10

ML: Machine Learning and Agents

Distributed and Incremental Learning with Agents and noisy Sensor Data

epubli, ISBN 9783746752228 (2018)

This Chapter outlines the challenges of machine learning in unreliable distributed environments and noisy input (sensory) data. The concept of learning agents is introduced used for distributed learning, e.g., in distributed sensor networks. Distributed agent-based learning is combined with incremental learning approaches to meet the requirements in evolving and dynamic environments.

Learning is closely related to the agent model implementing a generic artificial intelligent system using sensory data to plan and execute actions, shown in Figure *10.1*.

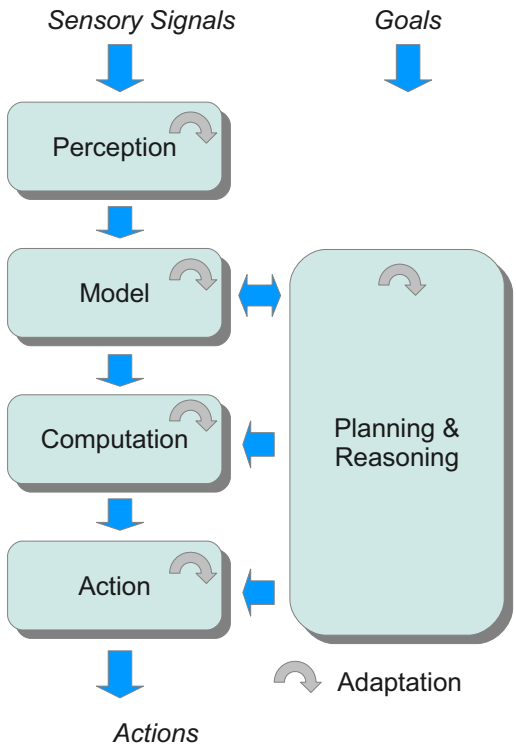


Fig. 10.1 *General Artificial Intelligence System with adaptivity based on learning closely related to the agent model*

10.1 Introduction to Machine Learning

An advantage of Machine Learning (ML) over numerical algorithms is the ability to handle problems having a Non-Polynomial (NP) complexity class, i.e., an exponential dependence of the computation time from the data size of the problem [WUE16].

Basically learning algorithms can be used to improve a system behaviour either at run-time or at design-time (or a hybrid approach of both) by optimizing function parameters to increase mainly estimation accuracy. The task of a Machine Learner is to derive a hypothesis (set) for a target concept, e.g., simply the Boolean variable *Overload* that gives the statement that there is probably an overload condition or not. Machine learning is based either on example data (training with labelled data) or on experience and history with reward feedback retrieved at run-time of a system. The labels are concrete values of the target concept (i.e., possible values of the output variable). The data consists of attribute variables, e.g., different strain gauge and temperature sensors.

The fields of application range from the optimizing and fitting of parameters of evaluation functions to full feature extraction classifiers. Machine learning is often used if there is no or only an incomplete world model specifying the output change of a module on behavioural or functional level (the system or a part of it) in response to a change of the input stimulus.

Machine Learning is closely related to the agent model suitable for the composition of complex system, as shown in Figure *10.2*. Agents can be considered as learning instances, supporting distributed learning.

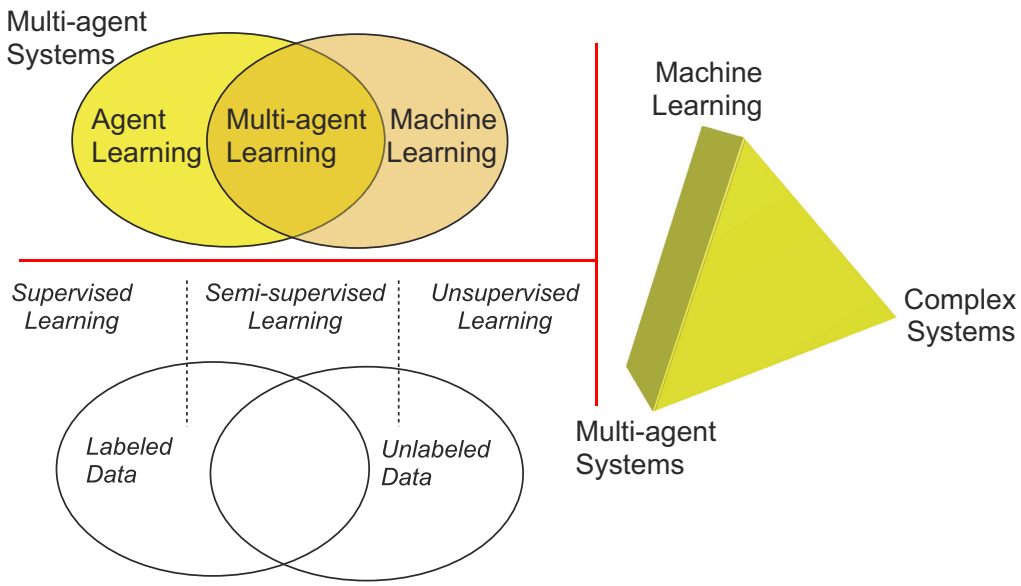


Fig. 10.2 *Supervised and unsupervised learning with relation to agent-based learning. Multi-agent-based learning can be supervised and unsupervised.*

A practically orientated classification of machine learning schemes (based on [SAM11]), suitable for information extraction in sensor networks and information retrieval is given by the following list:

Supervised Learning

A process that learns a model, basically a function mapping input data (of the past) to an output data set using data comprising examples with an already given mapping, i.e., using labelled data (see Figure *10.2*). That means the supervised learning task has to find a relationship between input attributes (independent variables) and output attributes (dependent variables). Two typical examples are classification and regression. Supervised learning produces a data base requiring a high demand of storage resources, but the resulting learned model can be considerable small (e.g., a decision tree).

Unsupervised Learning

This is generally a process that seeks to learn structure of data (see Figure *10.2*) in the absence of an identified output (like in supervised learning with labelled data) or a feedback (reinforcement learning). That means the unsupervised learning has to find distributions of instances by grouping instances without pre-specified dependent attributes. Examples are clustering or self-organizing maps trying to group similar unlabelled data sets.

Semi-supervised Learning

This is the combination of supervised and unsupervised learning techniques creating a hybrid learning architecture (see Figure *10.2*).

Reinforcement Learning

In some situations, the output of a system (the impact of the environment in that the system operates) is a sequence of actions, and the sequence of correct actions is important, rather than one single action. Reinforcement learning seeks to learn a policy mapping states to actions that optimizes a received reward (the feedback), finally optimizing the behaviour of a system (the reactivity). It is different from data classifiers and relates closely to the autonomous agent behaviour model. There are no trained example situations for correct or incorrect behaviours, only rules evaluating and weighting actions and their impact on the environment and the system.

Association Learning

The goal of association learning is to find conditional probabilities of association rules between different items of data sets. An association rule has the form $X \rightarrow Y$, where X and Y are item sets. The association rule belongs to a conditional probability $P(Y|X)$ giving the probability that if X occurs,

then set *Y* is also likely to occur. Giving user-specified support and confidence thresholds, the a-priori algorithm developed by [AGR96] can find all association rules between two sets *X* and *Y*. This proposed algorithm can be parallelized.

Classification

Classifier systems can be considered as modelling tools. Given a real system without a known underlying dynamics model, a classifier system can be used to generate a behaviour that matches the real system. The classifier offers rules-based model of the unknown system. There is a very large number of classification algorithms, for example, commonly used, decision trees (i.e., C4.5 algorithm), instance-based learners (i.e., nearest-neighbour (NN) methods), support vector machines (basically linear classifiers), rule-based learners, neural networks, and Bayesian networks. The main purposes of classification in sensor networks are knowledge extraction and pattern recognition. For example, in [BOS13B], C4.5 and k-NN algorithms were used to classify a 18-dimensional strain-gauge sensor vector of a flat rubber plate to a (*F,X*) vector providing a strength classification of an applied load (*F*) with an estimation of the spatial position (*X*).

Decision trees as one outcome of a machine learning classifier have low computational resource requirements and can be implemented directly on microchip level, for example, used for energy management in [BOS11B].

Clustering

Clustering is basically an unsupervised learning with the goal to group data sets sharing similar characteristics in clusters automatically. The main goal is finding structure in the given set of data. A common clustering algorithm is the k-mean algorithm, which quantifies vectors and compute the distance between vector, finally grouping the nearest vectors in cluster segments (see, e.g., [BEL15]). A Self-organizing map is another well-known clustering algorithm.

Regression

Like classification, regression is a supervised learning approach. The goal is to learn an approximation of a real-valued function mapping an input data vector (real-valued input variables) on an output vector (the mean of response variables).

Machine Learning can be represented by a four-layer model [ROK15]:

L1. Application

Structural Health Monitoring, Fault Detection, Adaptation of control sys-

tems, Prediction, ..

L2. Tasks

Classification, Regression, Clustering, ..

L3. Models

Models representing the learning outcome: Decision Trees, Bayesian Networks, Artificial Neural Networks, Self-organizing Maps, ..

L4. Algorithms

C4.5, nearest neighbour k-nn, ..

Machine Learning is divided basically in two phases that can be executed off-line, on-line, and mixed:

1. Model derivation (the learning of a prediction or clustering model from known data); and
2. Application of the learned model (applying new unknown data to the learned model function to predict a class or to get a relation to a cluster).

In general, ML derives a hypothesis $h(\boldsymbol{x})$ of a model function $f(\boldsymbol{x})$: $\boldsymbol{x} \rightarrow \boldsymbol{y}$ using a learner function M: $\mathbb{D} \rightarrow h$ *with* $h \in \mathbb{H}$ and a training data set $\mathbb{D}$. The model function $f(\boldsymbol{x})$ is also known as the target concept. Basically, the model function (or its hypothesis) maps an input data vector $\boldsymbol{x}=(a_1,a_2..,a_n)$ on an output data vector $\boldsymbol{y}$ with $\|\boldsymbol{y}\| \ll \|\boldsymbol{x}\|$, i.e., the mapping function performs feature extraction by reducing data. Each $\boldsymbol{x}_i$ is a data instance of an instance set $\boldsymbol{x} \in \mathbb{X}$ given by its attributes, that are the independent feature variables, $(a_1,a_2..,a_n)$ and the dependent output target attribute $\boldsymbol{y}_i$. In supervised learning a set of labelled data sets are used as training data, i.e., $\mathbb{D}=\{(\boldsymbol{x}_1,l_n),(\boldsymbol{x}_2,l_m),..\}$. In the case of classification, the set of labels $\mathbb{L}=\{l_1,l_2,..\}$ are possible values of the scalar output variable y, i.e., $y \in \mathbb{L}$. The general problem of the learning task is the diversity of different hypothesis functions $h \in \mathbb{H}$ that can be derived from training sets, approximating the model function more or less accurately. There are more general hypothesis functions and more specific. The learning task has to find an appropriate hypothesis from the hypothesis space $\mathbb{H}$.

Each learning task incorporates three different steps using different kind of input data:

Known Training Data

Task Learning: Find hypothesis $h(\boldsymbol{x})$ for unknown target function $f(\boldsymbol{x})$: $\boldsymbol{x} \rightarrow \boldsymbol{y}$

Known Test Data

Task Testing: Test hypothesis $h(\boldsymbol{x})$ and check quality and generalization of hypothesis

Unknown Data

Task Application: Apply hypothesis function $h(\boldsymbol{x})$ to unknown data

Feedback Data[1]

Task Feedback: Adapt the current hypothesis $h(\boldsymbol{x}) \rightarrow h'(\boldsymbol{x})$ with feedback (reward) from application providing new training data at run-time.

There are various different ML algorithms providing different learning approaches (i.e., learner function M) and representations of the learned model (i.e., hypothesis function h). They differ in:

- Appropriate matching of specific use cases (kind of input data and input data distribution)
- Accuracy (prediction quality)
- Speed: Computational complexity and Real-time Capability (with respect to learning and application)
- Storage requirements
- Distributivity (with respect to efficiency and communication costs)
- Adaptivity (i.e., support of incremental learning at run-time, if any)
- Noise immunity (Impact of input data noise on classification and prediction quality)

The deployment of ML using low-resource embedded systems can be a challenge with respect to computational complexity and storage requirements. The deployment of ML in distributed sensor networks providing inherent distributed input data demands for distributed ML algorithms operating on localized data and incremental on-line learning capabilities operating on data streams. In the following section a selection of ML algorithms are introduced suitable for the deployment on embedded systems and computational distribution. Probabilistic learning approaches (i.e, Bayesian Networks) are not discussed here as there is practically less or incomplete knowledge about probabilities of heterogeneous sensor data from various sources required for probabilistic approaches.

10.2 Decision Trees

Decision trees are attractive models representing a learned target concept as it can be read and understand easily. After a model is generated, it can be reported back to other people. Decision trees are commonly used for classification tasks only.

1.Optional

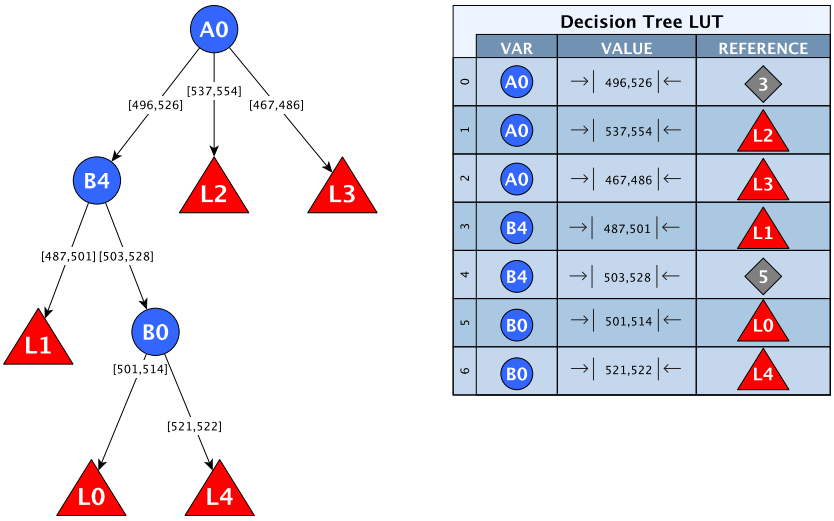


Fig. 10.3 *A learned decision tree and its linear look-up table representation (A_i,B_i: Attribute Variables, L_i: Class Symbol (label) i, n_i: Reference to a table row i)*

A decision tree (DT) is a directed acyclic graph consisting of nodes, leaves, and conditional edges. Each node of the DT is associated with an attribute variable and the edges are related to attribute values selecting an evaluation path. A leave is associated with a value of the target attribute (class symbol or numerical value). The main advantage of DTs compared with other models are their low storage requirement and the capability to store DTs in table format, shown in Figure *10.3*. A learned DT model does not carry any original training data.

The general task of DT learning is the selection of suitable attribute variables based on training data sets splitting a tree node in sub-trees leading to reliable classification paths along different nodes with high classification probability and diversity. A decision tree is built top-down from a root node and performs partitioning the data into subsets containing instances with similar values and creating sub-trees.

Algorithms - Overview

ID3

ID3 by J. R. Quinlan is an iterative algorithm for constructing DT from training data set $\mathbb{D}$. $\mathbb{D}$ is a table with columns (a_1,a_2,..,a_n,y) and a number of rows. Each table cell has a value $v \in V(a)$ (of attribute a) or $t \in T$ for the target attribute.

a_1	a_2	..	a_n	y
v_1	..	..	..	t_1
v_2	..	..	..	t_2
v_2	..	..	..	t_1
v_1	..	..	..	t_3
v_3	..	..	..	t_1
..	..	..	..	..

The node splitting attribute is selected based on gain and information entropy calculation.

Consider a column of the table, e.g., the target attribute column of *y*. There is a finite set of unique values $V=\{v_1,v_2,..v_m\}$ that *y* (and any column *c*) can hold. Now it is assumed that some values occur multiple times. The information entropy, i.e., a measure of disorder of the data, of this (or any other) column *c* (of attribute *a*/*y*) is defined by the entropy of the value distribution $entropy_N$:

$$entropy_N(N) = \sum_{i=1}^{n} -p_i \log_2 p_i, \text{with } p_i = \frac{N_i}{\sum N}$$
$$entropy(c) = \sum_{v \in V(c)} -p_v \log_2 p_v, \text{with } p_v = \frac{|c_v|}{|c|} \quad (10.1)$$

with p_v as the probability that a specific value *v* occurs in the column, i.e., the number of occurrences $n(v)=|c_v|$ of *v* in the column *c* relative to the number $n=|c|$ of rows in *c*. For example, the column *y* with 10 rows is occupied by two different values {A,B,B,A,A,A,B,B,B,B}, e.g., *y* relates to "Playing Golf" and A=Yes and B=No. Assume a distribution {4,6} of the set {A,B}. Then the entropy of *y* is

```
entropyN({4,6})=-4/10log(4/10)-6/10log(6/10) = 0.97.
```

Assume the attribute a_1 is related to the weather outlook with three possible values {Sunny, Overcast, Rainy}. The column of a_1 of the data set *D* contains the distribution {3,6,1}. Then the entropy of the column of a_1 is $entropy_N(\{3,6,1\})= 1.3$.

The calculation of the entropy of single columns is unhelpful for the selec-

tion of appropriate attributes. Moreover, an input attribute column must be related to the target attribute distribution. Now assume the outcome of the target variable $y \in \{A,B\}$ (in this example PlayingGolf) depends on the input attribute $a_1 \in \{\texttt{Sunny}, \texttt{Rainy}, \texttt{Overcast}\}$ (i.e. Weather). The column a_1 should contain 2(y=A) and 1(y=B) rows with a_1:Weather=Sunny, 4(y=A) and 0(y=B) rows with a_1:Weather=Overcast, and 1(y=A) and 2(y=B) rows with a_1:Weather=Rainy. Then the entropy of the column a_1 related to y is:

```
P(Sunny) entropyN({2,1}) + P(Overcast) entropyN({4,0}) +
P(Rainy) entropyN({1,2}) = 3/10*0.91+4/10*0.0+3/10*0.91 = 0.55
```

In general, the information entropy of a column $c(a)$ value distribution related to the outcome of the target variable is given by:

$$Entropy(T,c) = \sum_{v \in V(c)} p_v entropy_N(\{c_v \mid T\}),$$
$$\text{with } p_v = \frac{|c_v|}{|c|} \text{ and } \{c_v \mid T\} = \{|\, c_v \text{ with y=}t_1 |, |\, c_v \text{ with y=}t_2 |, ..\}, t_i \in T \tag{10.2}$$

with $v \in V$(a) as the possible unique values of the attribute variable a and T the values of the target attribute.

a/y	t_1	t_2	..	t_o	***Distribution***
v_1	n_{11}	n_{12}	..	n_{1o}	$\{n_{11},n_{12},..,n_{1o}\}$
v_2	n_{21}	n_{22}	..	n_{2o}	..
v_3	n31	n_{32}	..	n_{3o}	..
..	..	..	..	..	..
v_m	n_{m1}	..	..	n_{mo}	$\{n_{m1},n_{m2},..,n_{mo}\}$

The *ID3* algorithm now starts with an empty tree and the full set of attributes $A=\{a_1,..,a_n\}$. The entropy for each column is calculated (applying Equation *10.1* and *10.2*) and finally the information counting gain for each column with respect to the target attribute distribution T in this column:

$$Gain(T,c) = entropy(T) - Entropy(T,c) \tag{10.3}$$

The column c_i with the highest gain associated with attribute a_i is selected for the first tree node and removed from the set A. For each value of the attribute occurring in the selected column a new branch of the tree is cre-

ated (i.e., each branch contains the rows that has one of the values of the selected attribute). In the next iteration a new attribute (column) is selected from the remaining attribute set until there are no more attributes. A zero gain indicates a leave (i.e., all rows select the same target attribute, i.e., $T=\{t\}$).

C4.5

A major limitation of *ID3* is the creation of small and efficient DT if there are many different values of feature variables, e.g., real.valued data and target variables. Furthermore, *ID3* cannot handle undefined attribute values.

Instead, using the information gain for a split criteria given by Equation *10.3*, the *C4.5* algorithm considers the gain ratio that defines a relation of the information gain of a data column to the outcome of a possible splitting by creating a tree node (i.e., the split information entropy of the target attribute distribution) [HSS14].

This is given by:

$$\begin{aligned} Gain(T,c) &= entropy(T) - Entropy(T,c) \\ SplitInfo(T) &= \sum_{i=1}^{n} -\frac{|T_i|}{|T|}\log_2\frac{|T_i|}{|T|} \\ GainRatio(T,c) &= \frac{Gain(T,c)}{SplitInfo(T)} \end{aligned} \tag{10.4}$$

Like in the *ID3* algorithm, the data table is sorted at every node creation with respect to the best splitting attribute based on the gain ratio impurity. At each node of the tree, *C4.5* chooses one attribute of the remaining attribute set *A* that most effectively splits the remaining training data set into subsets enriched in one class or the other. The criterion is the normalized information gain (difference in entropy) that results from choosing an attribute for splitting the data. The attribute with the highest normalized information gain is chosen to make the decision.

In addition, the *C4.5* learner performs tree pruning to optimize the application of the learned model. One major flaw of *ID3*/*C4.5* is over-fitting of the tree and sub-tree replication. Furthermore, even *C4.5* does not handle noisy data accurately.

The computational complexity of the *C4.5* algorithm is about $\Theta(mn^2)$ [SU06] with m as the number of training examples and n as the number of attributes.

INN

The ε-interval nearest-neighbour algorithm (*INN*) uses a modified and simplified *ID3*/*C4.5* algorithm handling significantly noisy data combined with the NN approach used by the tree evaluation of the DT to optimize classification. It is well suited for noisy sensor data, geometrically correlated sensor data, and in advance unknown or incomplete training data sets like in incremental learning. Instead, using single discrete data values, data value intervals are used considering uncertainty. In the following the algorithm is introduced as a major algorithm suitable for the deployment in sensor networks.

Traditional Decision Tree Learner (DTL) (e.g., using the *ID3* 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 of a specific attribute variable), which is well suited for non-metric symbolic attribute values, like colour names, shapes, and so on. The distinction probability of two different symbols is usually one. In contrast, real-valued sensor data is noisy and underlies variations (e.g., drift) due to the measuring process and the physical world. Two numerical (sensor) values *A* and *B* have only a high distinction probability if the uncertainty intervals $[A-\sigma,A+\sigma]$ and $[B-\sigma,B+\sigma]$ 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 optimal data set splitting, a column ε-interval entropy computation is introduced, that extends each value of a column vector (associated to a specific attribute) with an uncertainty interval $[v_i-\varepsilon,v_i+\varepsilon]$, based on the *C4.5* algorithm.

Initially, the values of a real-valued data table is transformed in data range values:

a_1	a_2	..	a_n	y
$[v_1\pm\varepsilon]$	$[v_1\pm\varepsilon]$	..	..	t_1
$[v_2\pm\varepsilon]$	$[v_2\pm\varepsilon]$	..	..	t_2
$[v_1\pm\varepsilon]$	$[v_1\pm\varepsilon]$	..	..	t_1
$[v_3\pm\varepsilon]$	$[v_3\pm\varepsilon]$	..	..	t_3
..	..	..	..	t_1
..	..	..	..	..

Applying the 2ε-interval approach to original *ID3* algorithm from Equation

10.2 gives:

$$
\begin{aligned}
&Entropy(\varepsilon,T,c) = \sum_{v \in V(c)} p_\varepsilon(v,c,\varepsilon)\, entropy_\varepsilon(\{c_v \mid T\},\varepsilon)\\
&entropy_\varepsilon(c,\varepsilon) = \sum_{v \in V(c)} -p_\varepsilon(v,c,\varepsilon)\log_2 p_\varepsilon(v,c,\varepsilon)\\
&p_\varepsilon(v',c,\varepsilon) = \frac{\sum_{v \in V(c)} \begin{cases} 1 : overlap([v-\varepsilon, v+\varepsilon],[v'-\varepsilon, v'+\varepsilon]) \\ 0 : otherwise \end{cases}}{|c|}\\
&overlap(v_1,v_2) = \begin{cases} true : (\lceil v_1 \geq \lfloor v_2 \wedge \lceil v_1 \leq \lceil v_2 \rceil \rceil \rceil \rceil \vee \\ \qquad (\lceil v_2 \geq \lfloor v_1 \wedge \lceil v_2 \leq \lceil v_1 \rceil \rceil \rceil \rceil) \\ false : otherwise \end{cases}
\end{aligned}
\tag{10.5}
$$

In contrast to the original *ID3/C4.5* algorithms calculating the target dependent data entropy for all columns for optimal attribute selection there is a simplified ε-interval data-centric algorithm calculating the entropy for a data set column only without considering the target attribute relationship, based on Equation *10.6*. This simplification reduces the computational complexity and enables the deployment of DT learning in embedded systems under real-time conditions and incomplete training sets as they are occurred in incremental (stream-based) learning.

$$
\begin{aligned}
&entropy_\varepsilon(c,\varepsilon) = \sum_{v \in V(c)} -p_\varepsilon(v,c,\varepsilon)\log_2(p_\varepsilon(v,c,\varepsilon))\\
&p_\varepsilon(v',c,\varepsilon) = \frac{\sum_{v \in V(c)} \begin{cases} 1 : overlap([v-\varepsilon, v+\varepsilon],[v'-\varepsilon, v'+\varepsilon]) \\ 0 : otherwise \end{cases}}{|c|}\\
&overlap(v_1,v_2) = \begin{cases} true : (\lceil v_1 \geq \lfloor v_2 \wedge \lceil v_1 \leq \lceil v_2 \rceil \rceil \rceil \rceil \vee \\ \qquad (\lceil v_2 \geq \lfloor v_1 \wedge \lceil v_2 \leq \lceil v_1 \rceil \rceil \rceil \rceil) \\ false : otherwise \end{cases}
\end{aligned}
\tag{10.6}
$$

The occurrence probability is calculated by counting overlapping 2ε intervals of data column values. Based on this calculation, the best attribute

(feature variable) of a column with the highest information entropy is selected for creating a new tree node. 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 value, 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 Algorithm *10.1*. It creates a graph based on node attribute selection using intervals, e.g. $x \in [500..540]$, 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. The choice of an appropriate ε value requires some statistical knowledge from a prior analysis of the data set.

Alg. 10.1 *INN algorithm, simplified version* [BOS16B]

```
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)
 1. Iterate the model tree until a result leaf is found.
 2. 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 classification function *classify* applies input data to the model by iterating paths along the learned decision tree. Each variable along its path is evaluated by finding the next edge in the tree. If a tree node has more than two edges and/or edges have overlapping value intervals, the classifier selects the best matching feature variable edge by finding the nearest neighbour for the current feature variable value from the set of feature value intervals.

The definition of a common ε parameter used in the INN algorithm applying 2e intervals to variable data is only useful if all variables have data values in equal interval, i.e., $x_i \in [v_0,v_1]$. This is the case for sensors of the same class in distributed sensor networks. If the sensors are different and the sensor values have relevant different dynamic ranges an x_i-individual ε_i value has to be defined and the Equation *10.6* has to be changed using individual ε_i values.

10.3 Artificial Neuronal Networks

Artificial neuronal networks (ANN) can be used for classification, regression, and clustering, i.e., supporting supervised and unsupervised learning methodologies. In contrast to a DT representing a strict hierarchical processing model an ANN represents a parallel processing model.

An ANN is composed of a set of simple processing units (nodes), called neurons that represent activation function $f(\boldsymbol{x})$: $\boldsymbol{x} \rightarrow y$, mapping one or more input variables $\boldsymbol{x}=(x_1,x_2,..)$ on an output variable, commonly posing a binary set of values.

The taxonomy of ANNs basically distinguishes the following classes and architectures [SIB12]:

- Feed forward NN
  - Single-layer perceptron.
    A single-layer perceptron consists of a single layer of output nodes. Input variables are directly mapped on output variables using weights. The sum of the weighted input variables is calculated for each node, $s=\Sigma x_i$. If the sum value s is about a threshold s_{thr}, the neuron "fires" and exposes the sum value (or any other fixed value) at the output, otherwise it exposes another deactivated value (e.g., 0 or -1). Neurons with this kind of activation function are also called McCulloch-Pitts neurons or threshold neurons.
  - Multi-layer perceptron.
    A multi-layer perceptron consists of multiple layers of computational nodes interconnected in a feed forward architecture, shown in Figure *10.4*. A learning algorithm have to adjust the weights of the network to optimize correct prediction that is more complex with multi-layer perecptrons than with single-layer percpetrons. One common technique is to define an error function comparing the values of the output variables with expected values and back propagating the errors to the input layers to adjust the weights. The weights of each single connection are adjusted to minimize the error function outcome by a small amount.
- Recurrent NN
  In contrast to feed forward networks with uni-directional data flow there are connections propagating output variables to previous layers causing a bi-directional data flow in RNNs. FNN propagate data linearly from input to output nodes, whereas RNN's back propagation introduces non-linear functional behaviour.
  - Single recurrent NN
  - Hopfield NN
- Stochastic NN
  - Boltzmann Machines
- Modular NN
  Module NN are composed of multiple independent NNs performing a sub-task and operating on a sub-set of input data connected by an

intermediate and moderating layer that isolates the independent NNs from each other.

Examples of modular NN are:

- Committee of machines (CoM)
- Associative NN (ASNN)

In multi-layer ANNs (see Figure *10.4*), the input layer connects to the outside world and the output layer stores the prediction results. All neurons between the input and output layers are hidden neurons. The transfer function $t_n(\boldsymbol{i})$: $\boldsymbol{i} \rightarrow o$ of a neuron n is commonly a logarithmic function mapping the input variables of the neuron (the incoming edges from other neurons or input variables) on the output variable (outgoing edge to other neurons or output variable).

The learning algorithm of an ANN performs training of the perceptrons in two ways:

1. By adjusting the weights of the interconnections;
2. By reconfiguring the interconnection network (adding or removing edges of the network graph).

The commonly used back propagation algorithm is summarized in Algorithm *10.2* that is well suited due to its ability to generalize well on a wide variety of problems. Back propagation of errors through a feed-forward neural network requires a deviation of the activation function.

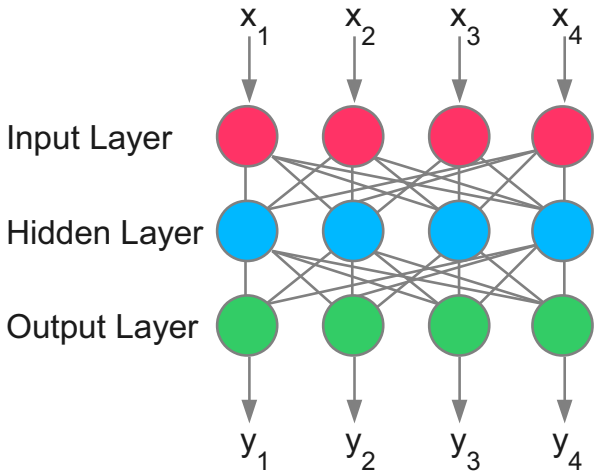


Fig. 10.4 *Principle architecture of a feed forward ANN consisting of three layers*

Alg. 10.2 *Back propagation algorithm used for ANN training*

```
1.      Initialize Network
2.      Set-up next input vector from the training data set
3.      Propagate input vector
4.      Calculate the error signal e=|y-y0|
5.      Propagate error signal to network
6.      Adjust weights to reduce error
7.      Repeat steps 2-5 to reduce error
        until the error is below a threshold e<ethr
```

10.4 Learning with Agents

The agent model poses some kind of autonomy and the capability to interact with an environment via perception and actions. Up to here only classification and regression tasks were considered. But adaptation of models at run-time is another major machine learning task. The agent architecture relies on the adaptation of the behaviour based on perception and reasoning. This adaptation can be performed by reinforcement learning.

According to [RAN07], learning and agent models can be combined by merging two operational cycles: The Agent Based Modelling (ABM) and the Machine Learning (ML) cycle, illustrated in Fig *10.5*, creating an incremental learning approach. From another point of view, the agents control the learning process. Such agent-based learning systems can be implemented, e.g., with ANNs posing multiple advantages and features:

- Fault Tolerance, i.e., missing or failing agents can be compensated by a self-organizing Multi-agent System;
- No base data assumption, i.e., composing and training ANNs do not require specific knowledge about input data distributions and statistics;
- "Organic Learning", i.e., ANNs and agents are not limited to prior given expert knowledge and can extend their knowledge at run-time;
- Incremental Learning, i.e., ANNs do not require initial extensive training;
- Learning of complex non-linear functions, i.e., ANNs can detect in principle all non-linear relationships between input and output variables.

Both cycles are similar. The integrated cycle inserts the ML cycle in the update block of the ABM cycle, i.e., updating the internal agent state based on the outcome of the learning cycle.

epubli, ISBN 9783746752228 (2018)

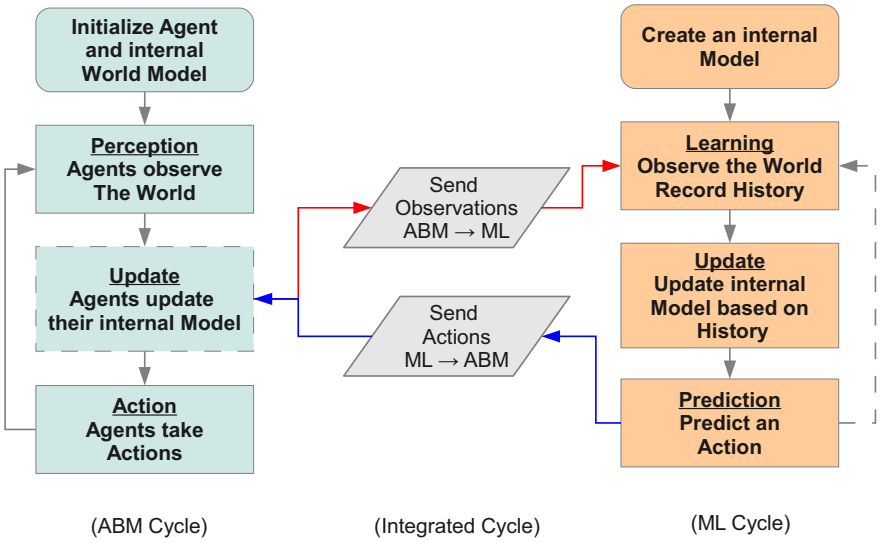


Fig. 10.5 *The composition of learning agents by merging the operational Agent Based Modelling and Machine Learning cycles (based on* [RAN07]*)*

Among using learning to adapt agents and multi-agent systems to environmental changes, agents can be used to perform adaptive learning for classification, regression, and clustering tasks in a cooperative manner and by using a divide-and-conquer approach, discussed in the next sections. For example, neurons of ANNs can be mapped on agents providing a dynamic reconfigurable network. Connections between neuron agents can be realized by simple directed communication.

10.5 Distributed Learning

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

Commonly sensing applications operate 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 resulting in

high power consumption. 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.

The machine learning algorithms presented in the previous sections are implemented as centralized instances. Furthermore, learning with training data is performed off-line with the entire training set and must be finalized before the learned model can be applied for classification of unknown data.

In distributed sensor networks the input data used for the learning task is inherently distributed and geometrically correlated. Instead collecting the entire sensor data by a central instance, a more useful and efficient approach is to distribute the learning algorithm by creating multiple distributed learning instances following this approach:

$$
\begin{array}{l}
M : D \rightarrow h(S) \\
l \in \boldsymbol{L} \\
h : S \rightarrow l \\
D : \{(S^1, l^1), (S^2, l^2), \cdots\} \\
S : \begin{pmatrix} x_{1,1} & \cdots & x_{n,1} \\ \vdots & \ddots & \vdots \\ x_{1,m} & \cdots & x_{n,m} \end{pmatrix}
\end{array}
\xrightarrow{\text{Distribution}}
\begin{array}{l}
m_{i,j} : d_{i,j} \rightarrow h_{i,j}(s) \\
h_{i,j} : s_{i,j} \rightarrow l_{i,j} \\
K : (l_{1,1}, l_{1,2}, \cdots) \rightarrow l \\
d_{i,j} : \{(s^1_{i,j}, l^1), (s^2_{i,j}, l^2), \cdots\} \\
s_{i,j} : \begin{pmatrix} x_{i-u,j-v} & \cdots & x_{i+u,j-v} \\ \vdots & \ddots & \vdots \\ x_{i+u,j-v} & \cdots & x_{i+u,j+v} \end{pmatrix}
\end{array}
\tag{10.7}
$$

with *M* as a global machine learner operating on global data *D*, *m* a local machine learner operating on spatially bound local data *d*. Each distributed machine learner *m* derives a hypothesis model *h* (predictor) operating on local data *d*. In contrast to a centralized predictor, there are multiple predictions that must be collected by a global instance *K* finally computing a global prediction based on the local ones, e.g., by using majority election of votes given by the localized predictors.

10.5.1 Event-based Sensor Processing

Distribution of learning in sensor networks requires an event-based, robust, and decentralized sensor data processing as the prerequisite for the ML task [BOS16B]. That means:

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-based learning system 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.

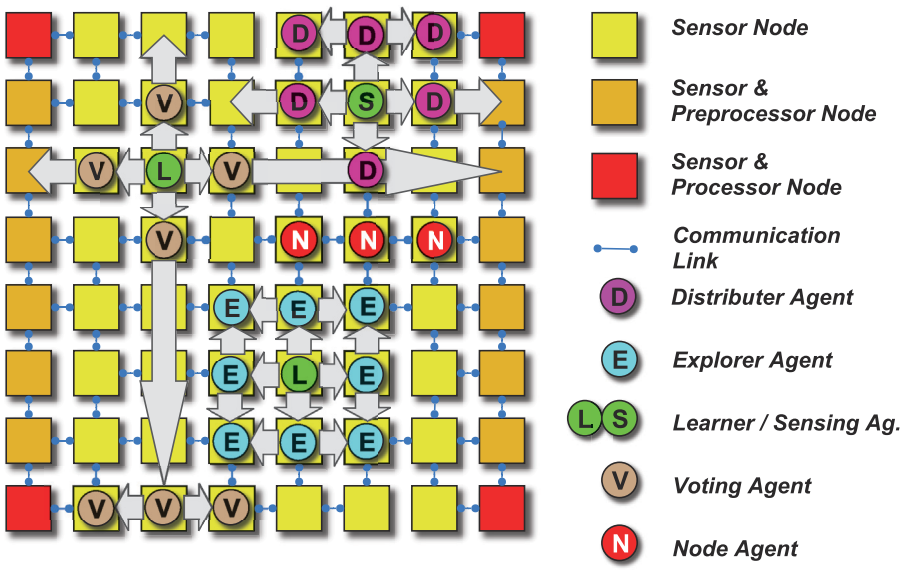


Fig. 10.6 *The logical view of a sensor network with a two-dimensional mesh-grid topology (left) and examples of the population with different mobile and immobile agents (right): node, learner, explorer, and voting agents.*

In many sensing applications like Structural Monitoring predicting a load situation or health conditions, sensor nodes are commonly arranged in some kind of two-dimensional grid network (as shown in Figure *10.6*). The sensor nodes provide spatially resolved and distributed sensing information of the surrounding technical structure, for example, a metal plate or a composite material with attached strain gauge sensors. Usually a single sensor cannot provide any meaningful information of the mechanical structure.

The sensor network can contain missing or broken links between neighbour nodes. Immobile node agents are present on each node (i.e., node service agents) performing sensor processing, agent control, and event detection. Node agents on 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.

Spatially bound regions in the network, Regions of Interest (ROI), are used to compute an event-based prediction and classification of the load case situation using supervised machine learning. Mobile agents are used to collect (percept) and deliver sensor data, but only limited to the ROI, shown in Figure *10.6* (explorer agents delivering neighbourhood sensor data to learner agents).

10.5.2 Distributed Learning Algorithm DINN

According to Equation *10.7* the learning and prediction task is divided into multiple independent learning instances operating on local data. Pattern recognition (sensor events) identify the aforementioned ROIs that activate learning instances (either for training or prediction). In the prediction phase a set of prediction results from individual learner instances are collected and a final prediction result is obtained by majority voting, summarized in Algorithm *10.3*.

Alg. 10.3 *Basic DINN algorithm: A multi process view communicating signal events*

```
global Process World:
  Create node processes {Node_1,..Node_N}

Process Node_i:
  Create processes: {sensor_i,learner_i}

Process sensor_i:
  If there is a sensor change of this S_i Then Create explorer agent

Process explorer_i:
  Explore the neighbourhood around the origin S_i,
    collect sensor signals {S_i-n,..S_i,..,S_i+n}.
```

```
  If there is a significant change of sensors in the neighbourhood
    Then Signal Event (SENSORS,Di={Si-n,..Si,..,Si+n})

Process learneri:
  Loop:
    Wait For Event (SENSORS,Di)
    If in mode Collecting Then Add new training data DS = DS + Di
    Else If in mode Learning Then
      M = INN.createtree(DS)
    Else If in mode Predicting Then
      Resulti = INN.classify(M,Di)
      Signal Event (VOTE,Resulti)

global Process election:
  Loop:
    Wait For Events (VOTE,Resulti)
    Make a voting decision: Majority wins!
    Signal Event (PREDICTION,Resultmost)
```

10.5.3 Distributed Learning with MAS

Figure *10.7* gives an overview of the composition of a complete sensor processing and distributed learning system with different agent classes. The MAS consists of the following agents:

- Node Agent
- Explorer Agent
- Sensing Agent
- Distributor Agent
- Notification Agents
- Learning Agent
- Voting Agent
- Election Agent

Some classes are super classes composed of sub-classes (e.g. the learner and the explorer class). A sensor node is managed by the 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).

The event-based regional learning leads to a set of local prediction 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.

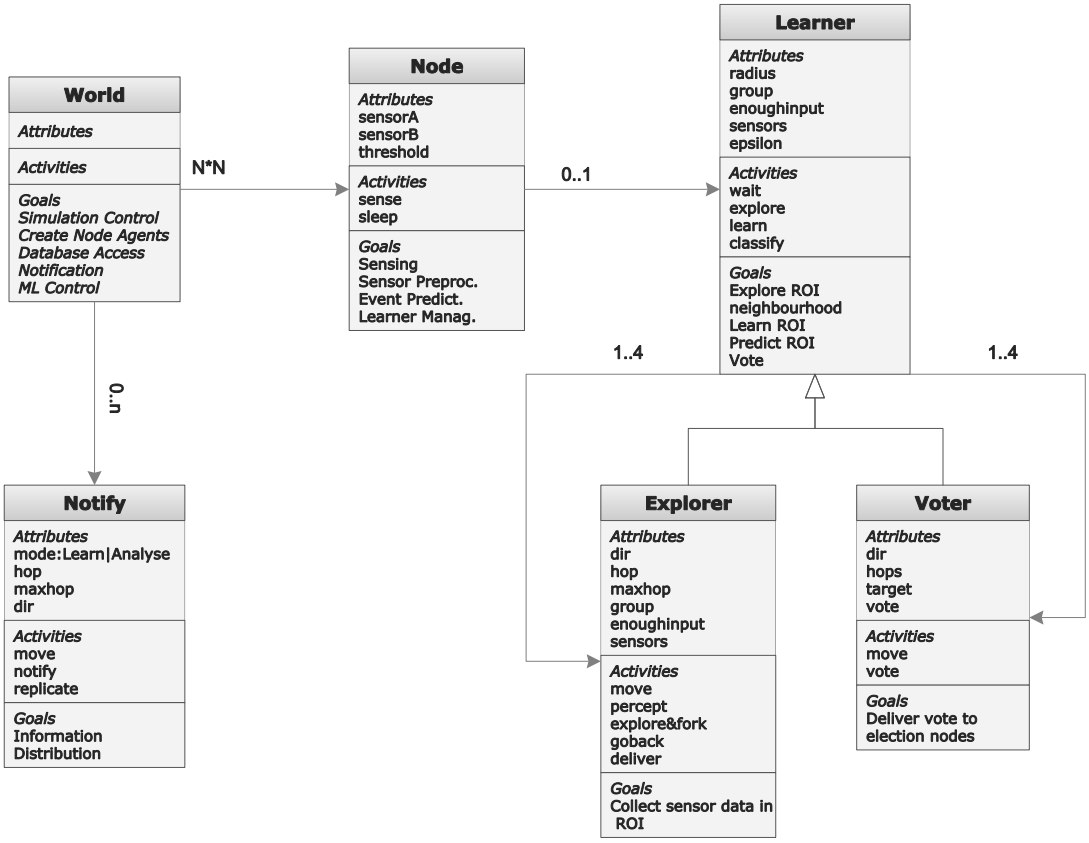


Fig. 10.7 *Overview of different agent classes and sub-classes used for the sensor processing and learning in the network and their relationships (grey solid arrow: agent instantiation at run-time, light arrow: sub-class relationship). The world agent is artificial and is only used in a simulation and handles the physical world and the network.*

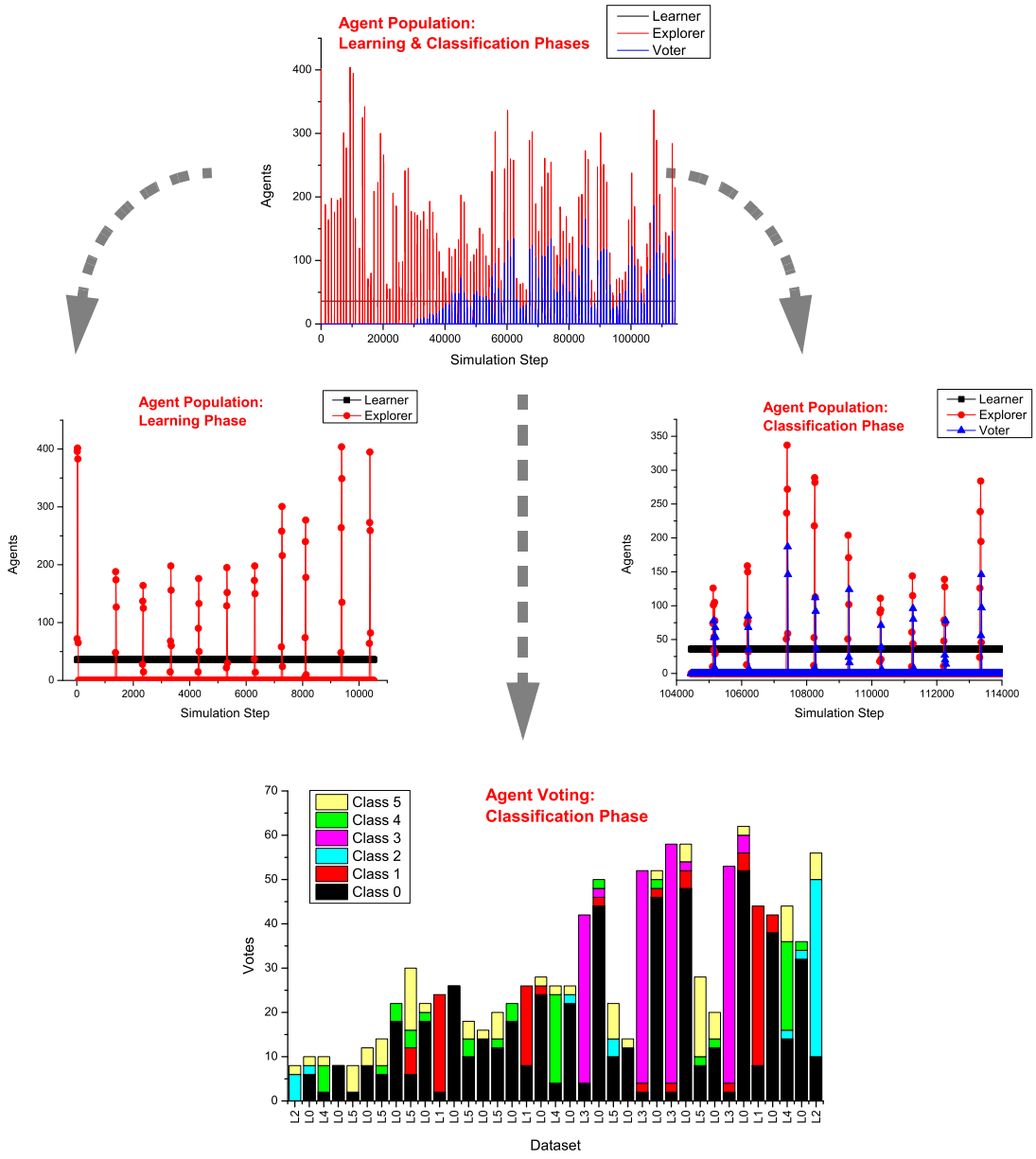


Fig. 10.8 *Simulation Results. The top figure shows the temporal agent population for a long-time run with a large set of single training and classification runs, with a zoom shown in the middle two figures. The bottom figure shows global classification results obtained by major voting of all event-activated regional learner agents.*

10.5.4 Distributed Learning: Case Study

To evaluate the distributed learning approach, an extensive MAS simulation was performed. The simulation assumes a spatially two-dimensional sensor network (see Figure *10.6* for details) with nodes arranged in a mesh grid connecting each node with up to four neighbour nodes. Each sensor node is attached to a strain gauge sensor used to measure strain of an artificial plate. The artificial sensor values were derived by inverse numerical computation and transferred to the MAS simulation. Some simulation results are shown in Figure *10.8*. The agent population plots show the efficient data processing of the event-based sensor processing and learning activities performed by the agents.

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.

10.6 Incremental Learning

The Machine Learning approaches presented in the previous section operate in two sequentially phases: (1) Learning (Deriving the prediction model with known training data) and (2) Application (Performing the classification using unknown data). Jiang [JIA13] showed that it is possible to perform incremental learning at run-time using trees, very attractive for agent and SoS approaches. A learned model (carried by the learner agent) is used to map data vectors (of an input variable set x_1,x_2,..; the feature vector) on class values (of an output variable y). The model can be updated at run-time by adding new training data or by updating the learned model by back propagation and reinforcement learning. The classification tree consists of nodes testing a specific feature variable, i.e., a particular sensor value, creating a path to the leaves of the tree containing the classification result, e.g., a mechanical load situation. Among the distribution of the entire learning problem, event-based activation of learning instances can improve the system efficiency significantly as shown in the previous section, and can be considered as part of the distributed learning algorithm (a pre-condition). Commonly the locally sampled sensor values are used for an event prediction, waking up the learner agent, which collect neighbourhood data by using a divide-and-conquer system with explorer child agents.

Combining the previously introduced distributed learning approach with incremental learning algorithms enables a self-adaptive learning system with a feedback loop, suitable for sensor processing, e.g., by integrated sensor networks in structural monitoring or by wide-area sensor networks.

epubli, ISBN 9783746752228 (2018)

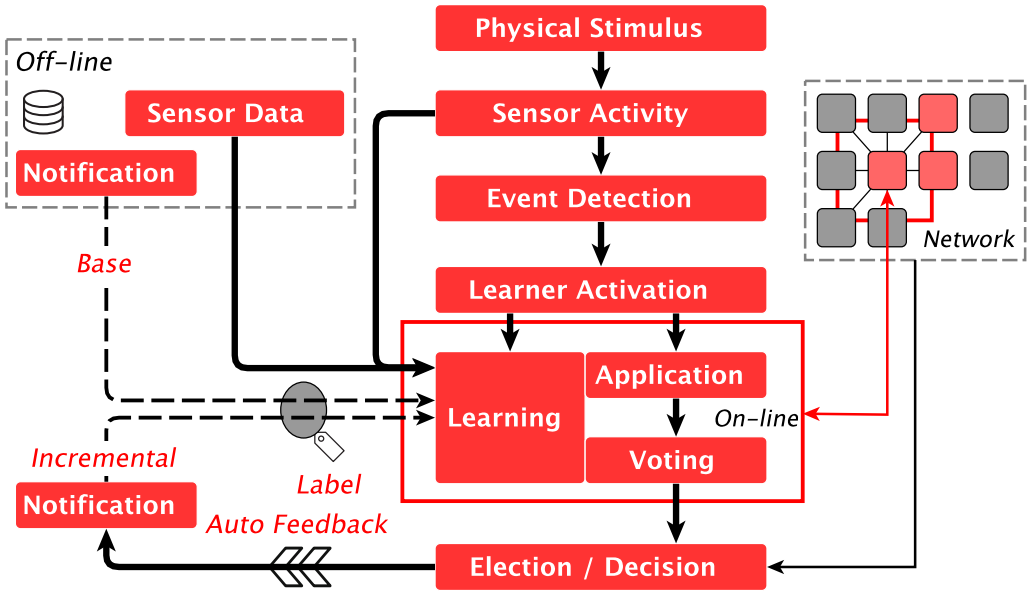


Fig. 10.9 *The principle concept: Global knowledge based on majority decision is back-propagated to local learning instances to update the learned model.*

The run-time behaviour flow of such a decentralized learning system is shown in Figure *10.9*.

Basically there are two different classes of Incremental learning algorithms: (1) A learned model is updated with new training data sets by adding the new sets to a stored database of old training sets; (2) A learned model is updated with new training data sets but without a data base of old sets.

Since agents should perform learning and classification in a distributed manner it is necessary to apply class (2) to minimize storage and communication complexity. An agent must store the training data and the learned model and it is useful to store only the learned model that can be updated at run-time without saving the entire history data.

10.6.1 Incremental Learning Algorithm I^2NN

The new algorithm for the incremental updating of a learned model (decision tree) with new training set(s) is shown in Figure *10.10* (in detail defined in Algorithm *10.4*). The initial model can be empty. The current decision tree can be weakly structured for a new training set (new target), i.e., containing variables unsuitable for separation of the new data from old one, which can result

in a classification of the new target with insignificant variables. Therefore, if a new node is added to the tree the last node is expanded with an additional strong (most significant) variable of the new data set (it is still a heuristic for future updates), i.e., creating an overdetermined tree.

The decision tree (DT) to be constructed consists of *Feature* and *Value* nodes, and *Result* leaves. First the current DT (model) is analysed. All feature variables x and their value bounds found in the tree in `Feature(x,vals)` nodes are collected in the `featureM` set (a list of (x, lower bound, upper bound) tuples). New feature variables added to the tree should not be contained in this set. Now each new training set, consisting of data variables x and a target result variable y, is applied to the DT. If the DT is empty, and initial *Feature* node is created using the most significant data variable As mentioned before, another *Feature* node is added for future DT updates. If the DT is not empty, the tree is iterated from the root node with the current training data until a feature variable separation is found, i.e., a new `Value(x,..)` node can be added with an non-overlapping 2ε-interval around the current value of the variable x. If the current 2ε-interval of a value of a feature variable overlaps an existing *Value* interval, this interval is expanded with the new variable interval and the DT is entered one level deeper. If the last Result leave is found and its value is not equal to the current target variable value, the update has failed.

This simple learning algorithm has a computational complexity of $\Theta(N)$ with respect to the number of training sets N to be added, and for each data set $\Theta(\log n)$ with respect to the number of nodes n in the current tree, assuming a balanced tree. The incremental learning is significantly simpler than the entropy-based feature selection *IDT* algorithm.

A physical stimulus results in sensor activity at multiple positions in the sensor network that is analysed by an event recognition algorithm (see previous section for an explanation). If a sensor node detects a local sensor event the local learner is activated. It performs either a learning of a new training set or applies the learned model with the current data set consisting of ROI data. In the case of a prediction, it will make a vote. After the election of all votes, the result is back propagated to the network and all learners can update their model with the current data set as a new training set.

epubli, ISBN 9783746752228 (2018)

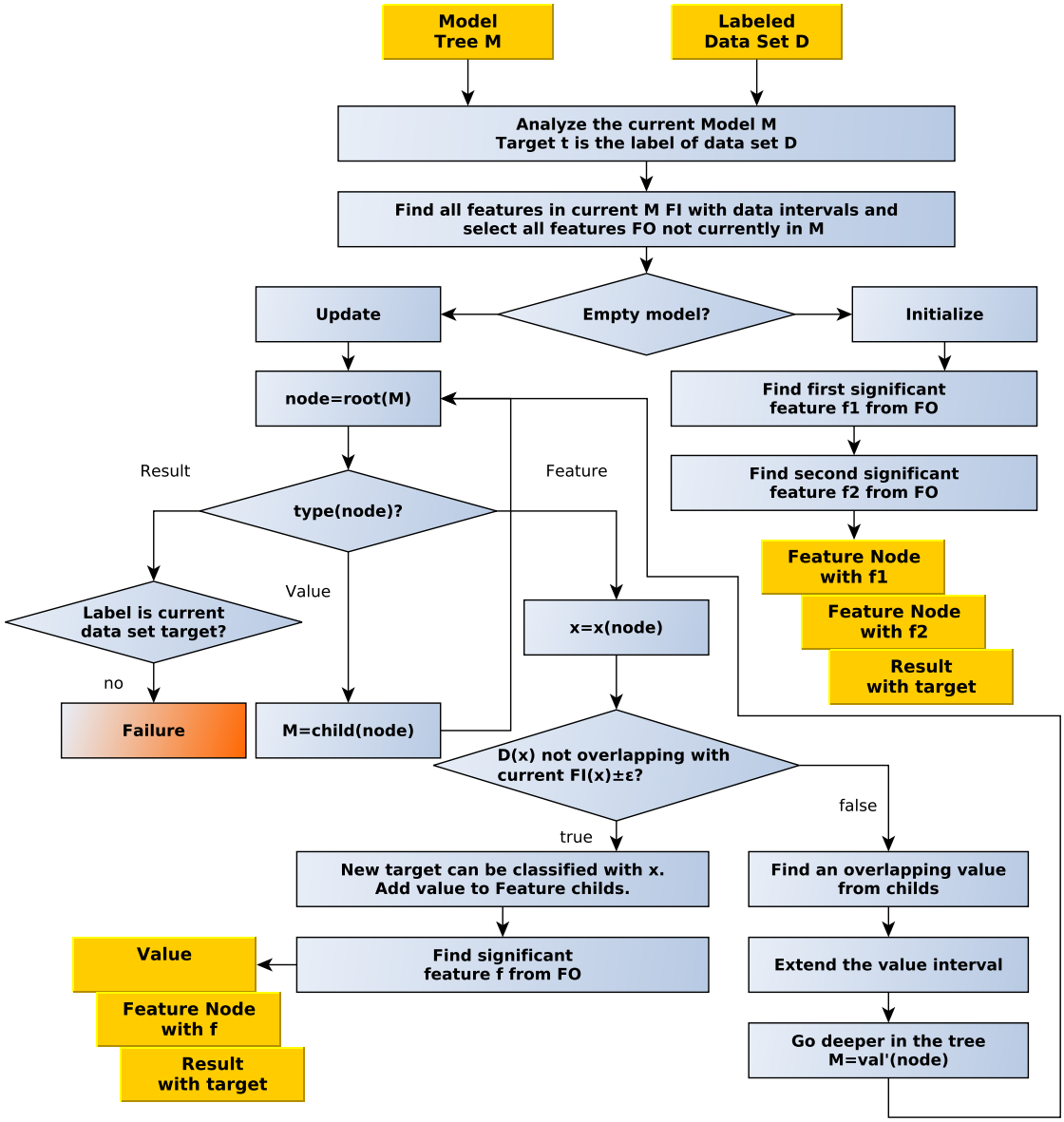


Fig. 10.10 *The processing flow of the new I²NN Algorithm*

Alg. 10.4 *Incremental Interval Decision Tree Learner Algorithm (I^2NN) (x: feature variable, y: output target variable, $\lfloor x$: lower bound of variable x, $\lceil x$: upper bound, v(x): value of x)*

```
1   type node = Result(t) |
2               Feature(x,vals:node []) |
3               Value(v,child:node)
4   type of dataset = (x1,x2,x3,..,y) []
5
6   function learnIncr(model,datasets,features,target,options) {
7     y = target
8     ε = options[ε]
9     Analyze the current model tree
10    featuresM = { (xi,⌊xi,⌈xi) | Feature(xi) ∈ model }
11
12    features' = { f ∈ features | f ∉ featuresM };
13    Create a root node
14    function init(model,set) {
15      f1 = significantFeature(set,features)
16      features' := { f | f ∈ features' ∧ f ≠ f1 }
17      f2 = significantFeature(set,features')
18      features' := { f | f ∈ features' ∧ f ≠ f2 }
19      featuresM := featuresM ∪ {(f1,v(f1)-ε,v(f1)+ε),(f2,v(f1)-ε,v(f2)+ε)}
20      model =
21       Feature(f1,[Value([v(f1)-ε,v(f1)+ε],
22         Feature(f2,[Value([v(f1)-ε,v(f2)+ε],
23           Result(v(y))]]
24    }
25    Iterate and update tree
26    function update(node,set,feature) {
27      when node is Result:
28       if t(node) ≠ y(set) then Failure!
29      when node is Feature:
30       x = x(Feature)
31       if set[x] not in ⌊featuresM(x)±ε⌉ then
32         New target can be classified
33         f1 := significantFeature(set,features')
34         featuresM := featuresM ∪ {(f1,v(f1)-ε,v(f1)+ε)}
35         Extend interval
36         ⌊featuresM(x) := min(⌊featuresM(x), set[x]-ε)
37         ⌈featuresM(x) := max(⌈featuresM(x), set[x]+ε)
38         leaf =
39           Value([set[x]-ε,set[x]+ε],
40            Feature(f1,[Value([v(f1)-ε,v(f1)+ε],
41             Result(v(y))]]
42         add leaf to vals(Feature)
43       else
44         Go deeper in the tree, find an
45         overlapping value and extend the interval
46         with val ∈ vals(Feature) | ⌊v(val)⌉ overlap ⌊set[x]±ε⌉ do
47           Extend interval
```

```
48          ⌊v(val) := min(⌊v(val), set[x]-ε)
49          ⌈v(val) := max(⌈v(val), set[x]+ε)
50          update(val,set,x(node))
51      when node is Value:
52        update(child(node),set)
53    }
54    Apply all new training sets
55    ∀ set ∈ datasets do:
56      if model = ∅ then init(model,set)
57      else update(model,set)
58
59    return model
60  }
```

10.6.2 Distributed Incremental Learning Algorithm DI^{2}NN

Like in the *DINN* algorithm, the *I^{2}NN* learning and prediction algorithm is distributed by applying local data to multiple learning instances and finally applying majority voting to predictions of multiple instances. Learning instances are activated by significant sensor changes (event-based learning and predicting).

10.6.3 Distributed Incremental Learning MAS

The Multi-agent System consists basically of the same agent classes used in the distributed non.incremental learning approach from the previous section.

In [BOS17C], a real sensor network consisting of seismic stations were transformed to a two-dimensional mesh-grid, placing station nodes based on a spatial neighbourhood relation to other stations, shown in Figure *10.11*. The sensor network is populated with different mobile and immobile agents. Non-mobile node agents are present on each node. Sensor nodes create learner agents performing regional learning and classification. Each sensor node has a set of sensors attached to the node, e.g., vibration/acceleration sensors. Agents interact with each other by exchanging tuples via the tuple space and by sending of signals. All agents were implemented in *AgentJS*, and allocate about 1k-10k Bytes for the entire process including code and data. Some agents, e.g., the explorer, is partitioned in a main and smaller sub-classes used only for the creation of child agents.

Node Agent

The immobile node agent performs local sensor acquisition and pre-processing:

- Noise filtering;
- Validation of sensor integrity;
- Sensor fusion;

- Down sampling of sensor data and storing data in tuple space;
- ROI monitoring by sending out explorer agents (optional) performing a correlated cluster recognition;
- Energy Management;
- Activation of learner agent.

Explorer Agent

The explorer agent is used to collect sensor data in an ROI by performing a divide-and-conquer approach with child agent forking. This approach provides robustness against communication failures and weak node connectivity.

- On the starting node, initially a set of explorer agents is sent out from to all possible directions.
- Each explorer agent migrates to the neighbour node, collect and processes local sensor data, and sends out further child explorer agents to all neighbourhood nodes except the previous node.
- All child explorer agents collect sensor data, divide themselves until the boundary of the ROI is reached, and return to the parent agent and deliver the collected sensor data. The approach is redundant, and hence multiple explorer agents can visit one node. The first explorer on a new node stores a marking in the tuple space, notifying other explorers to return immediately.
- After all explorer agents returned, the collected sensor matrix is delivered in the tuple space.

Learner Agent

The learner agent has the goal to learn a local classification model with data from an ROI. It can operate in two modes: (1) Learning (2) Classification (Application).

- The learner sleeps after start-up until it is woken up by the node agent. The synchronization takes place via the tuple space by consuming a `TODO` tuple.
- Learning mode: The `TODO` tuple contains the target variable value. The learner sends out explorer agents to collect the

sensor data (three sensors *HHE*, *HHN*, *HHZ*) in the ROI.

- If it uses the IDT algorithm, the learner stores the data set in its own data base. The current training data base is used to learn the model.
- If it uses the I^2NN algorithm, the learner only updates the current model and discards the current training data.
- Application mode: The learner sends out the explorer agents to collect sensor data in the ROI. It uses this sample data for prediction. If the classification was successful, it will send out voter agents.

Distributor Agent

Among the local learning approach, sensor data is collected by central instances, e.g., performing model-based seismic data analysis. A distributor agents is used to deliver a set of sensor data from an ROI. The distributor agent is activated only if there was a significant sensor change in the ROI. The distributions process can be performed with different approaches:

- Peer-to-peer, i.e., the destination node is known or a path to the destination must be explored.
- Broadcasting, i.e., using divide-and-conquer with agent replication.
- Data sink driven, i.e., the distributor agent follows a path of marking (stored in the tuple space) to deliver the data along this path.

Voter Agent

The voter agent distributes a particular vote to election agents. There can be multiple election agents in the network, hence a row-column network distribution approach is used.

- The initial node agent sends out multiple voter agents to all possible directions (four directions in a mesh-grid network).
- If a distributor agent reaches a boundary of the network, it will replicate and distribute the vote in perpendicular and opposite directions until a node with an election agent is found.

Election Agent

An election agent is some kind of central instance of the MAS.

- Collecting of votes delivered by voter agents within a time in-

terval (after that votes are discarded)

- Back propagation of winner votes to the learner agents by sending out notification agents.

Notification Agent

The notification agents are sent out by some central instance to notify all nodes that a new training set is available with a specific target variable value, e.g., the parameters of an earthquake event (identifier, location, magnitude,..). One central instance can be the election agent that evaluate votes in application mode. The winner vote fraction is carried by notification agents to update learner agents.

- A divide-and-conquer approach with agent replication is used to broadcast the notification to all nodes in the network.

Disaster Management Agent

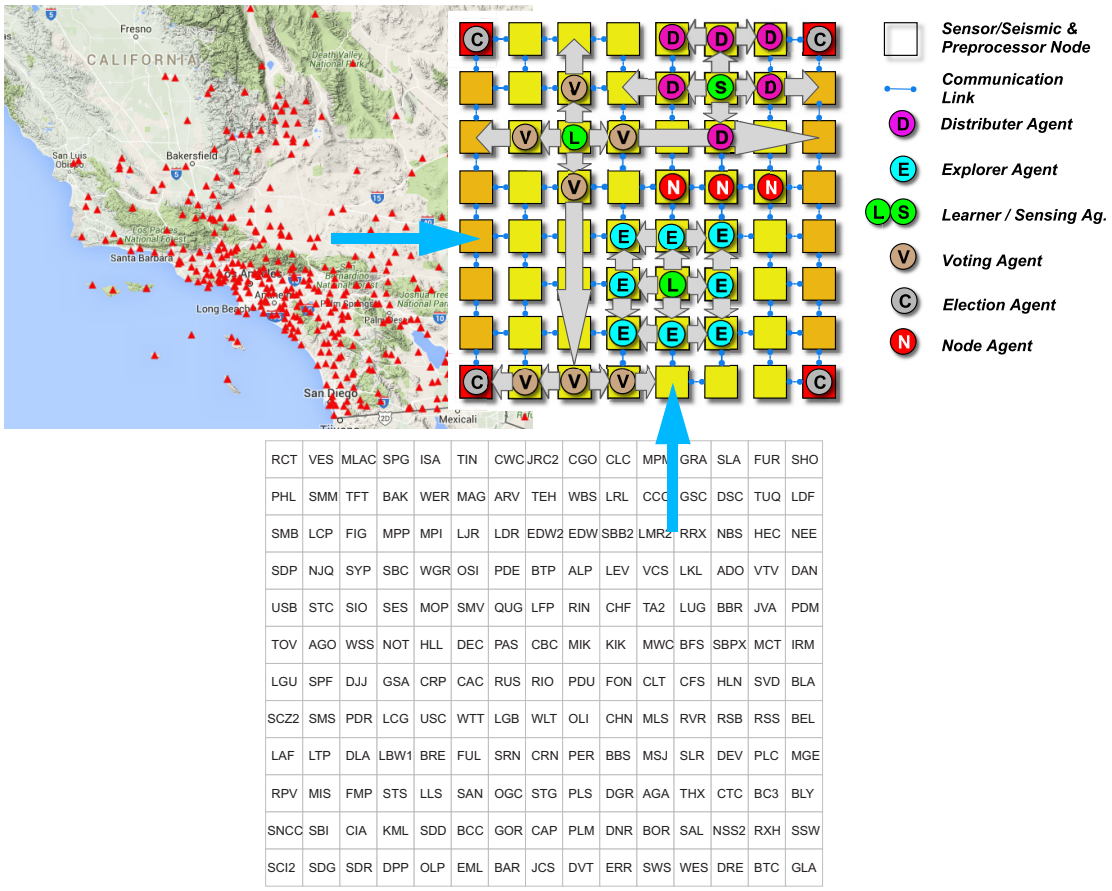
Although not considered in this work and currently not existing in the MAS, disaster management agents are high level central instances in the network that monitor the election results and activity in the sensor network and plan the co-ordination of disaster management based. See [FIE07] for a consideration of MAS-based disaster management.

10.6.4 Distributed Incremental Learning: Case Study

In [BOS17C] the MAS simulation was performed with data from a seismic network to evaluate the incremental learning approach. The real existing seismic network used for monitoring earthquake events was mapped on a two-dimensional mesh network, shown in Figure *10.11*. Each node of the network is considered as sensor or computational node representing an agent processing platform, which can be populated with mobile and immobile agents. Station data from different earthquake events were used to compute the sensor stimulus.

The seismic input data is high-dimensional. Therefore, a major challenge is data reduction. The original test data contains temporal resolved seismic data of at least three sensors (horizontal East, horizontal West, and vertical acceleration sensors) with a time resolution about 10ms, resulting in a very high-dimensional data vector.

Usually a seismic sensor samples only noise below a threshold level, mainly resulting from urban vibrations and sensor noise itself. For machine learning, only specific vibration activity inside a temporal Region of Interest (ROI) is relevant.



RCT	VES	MLAC	SPG	ISA	TIN	CWC	JRC2	CGO	CLC	MPM	GRA	SLA	FUR	SHO
PHL	SMM	TFT	BAK	WER	MAG	ARV	TEH	WBS	LRL	CCC	GSC	DSC	TUQ	LDF
SMB	LCP	FIG	MPP	MPI	LJR	LDR	EDW2	EDW	SBB2	LMR2	RRX	NBS	HEC	NEE
SDP	NJQ	SYP	SBC	WGR	OSI	PDE	BTP	ALP	LEV	VCS	LKL	ADO	VTV	DAN
USB	STC	SIO	SES	MOP	SMV	QUG	LFP	RIN	CHF	TA2	LUG	BBR	JVA	PDM
TOV	AGO	WSS	NOT	HLL	DEC	PAS	CBC	MIK	KIK	MWC	BFS	SBPX	MCT	IRM
LGU	SPF	DJJ	GSA	CRP	CAC	RUS	RIO	PDU	FON	CLT	CFS	HLN	SVD	BLA
SCZ2	SMS	PDR	LCG	USC	WTT	LGB	WLT	OLI	CHN	MLS	RVR	RSB	RSS	BEL
LAF	LTP	DLA	LBW1	BRE	FUL	SRN	CRN	PER	BBS	MSJ	SLR	DEV	PLC	MGE
RPV	MIS	FMP	STS	LLS	SAN	OGC	STG	PLS	DGR	AGA	THX	CTC	BC3	BLY
SNCC	SBI	CIA	KML	SDD	BCC	GOR	CAP	PLM	DNR	BOR	SAL	NSS2	RXH	SSW
SCI2	SDG	SDR	DPP	OLP	EML	BAR	JCS	DVT	ERR	SWS	WES	DRE	BTC	GLA

Fig. 10.11 *(Top, Left): The South California Seismic Sensor Network CI [Google Maps] (Top, Right) Sensor Network with stations mapped on a logical two-dimensional mesh-grid topology with spatial neighbourhood placing, and example population with different mobile and immobile agents. (Bottom) The station map.*

To reduce the high-dimensional seismic data, (I) The data is down sampled using absolute peak value detection; (II) Searching for a potential temporal ROI; and (III) Down sampling the ROI data again with a final magnitude normalization and a 55-value string coding.

The process is shown in Figure *10.12*. The compacted 55-string coding assigns normalized magnitude values to the character range `0`, `a-z`, and `A-Z` (`!`), with `0` indicating silence, and `!` overflow. If there were multiple relevant nearby vibration events separated by "silence", a * character separator is inserted in the string pattern to indicate the temporal space between single patterns.

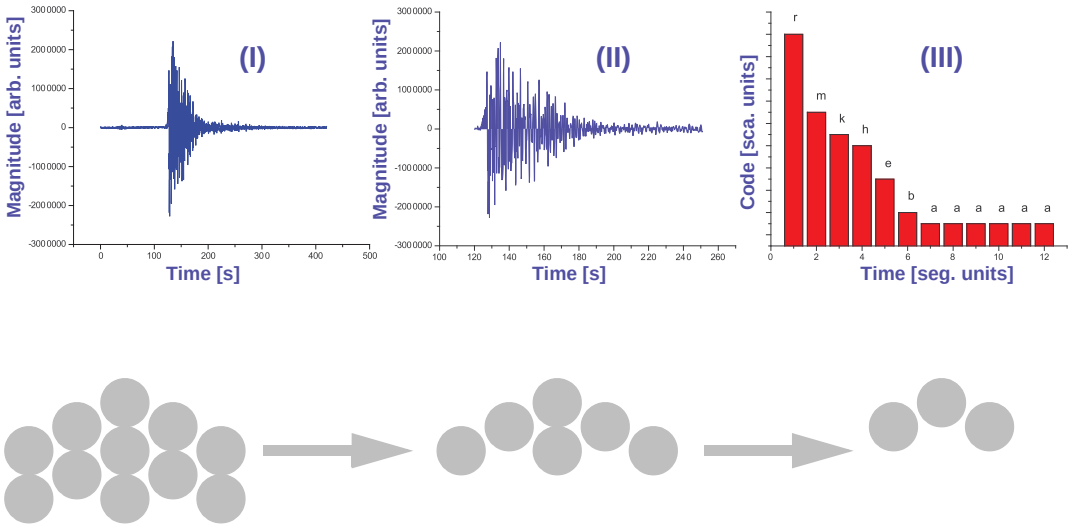


Fig. 10.12 *Data reduction of high dimensional temporal sparse data: (I) Down sampling (1:16) with absolute peak value detection, (II) ROI analysis and ROI clipping, (III) Down sampling (1:64) and scaling/normalization with 55-string coding (0,a-z,A-Z,!,*)*

The vibration (acceleration) is measured in two perpendicular horizontal and one vertical directions. This gives significant information for an earthquake recognition and localization.

The data reduction is performed by a node agent present on each seismic measuring station platform. Only the compact string patterns are used as an input for the distributed learning approach. Based on this data, the learning system should give a prediction of an earthquake event and a correlation with past events. To deploy regional learning for a spatial ROI, seismic stations should be arranged in a virtual network topology with connectivity reflecting spatial neighbourhood, e.g., by arranging all station nodes in a two-dimensional network. The virtual links between nodes are used by mobile agents for exploration and distribution paths. They do not necessarily reflect the physical connectivity of station nodes.

The evaluation of the distributed incremental learner is performed by applying the data of different earthquake events to the distributed system in a random sequence. The learned prediction model should classify earthquake events and should recognize similar events for disaster management. The distributed incremental learning algorithm *DI*^{2}*NN* is compared with results from non-incremental learning (using the distributed *DINN* algorithm).

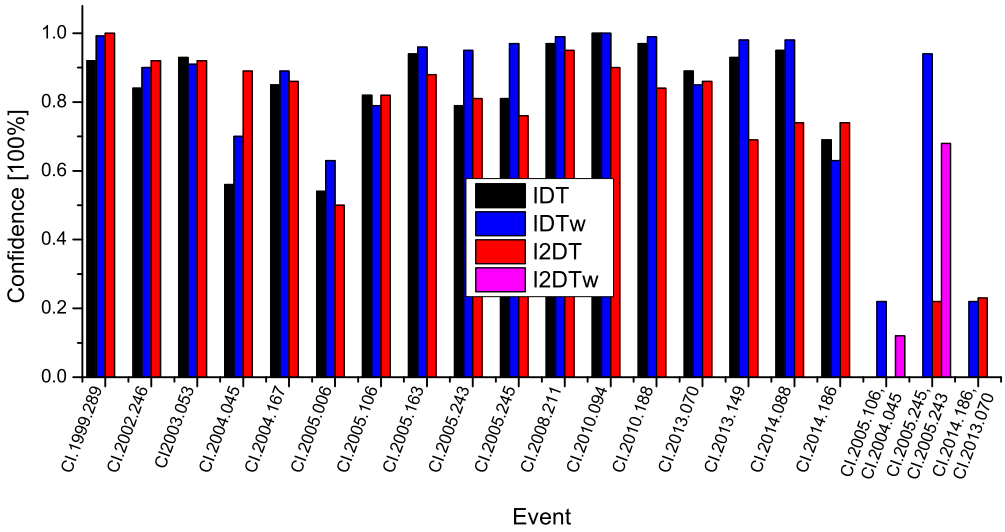


Fig. 10.13 *Comparison of distributed non-incremental (IDT≡DINN) with incremental (I2DT=DI²NN) learning and additional comparison of weighted (w) and not weighted votes using same training data* [BOS17C]

Figure *10.13* shows obtained results from simulation that poses a high prediction quality of the incremental learner compared with the non-incremental learner.

All predictions based on global majority election and Monte Carlo simulation. Multiple learning runs were performed to train the network using a random sequence of different earthquake events with noisy data (complete set). During the classification (application) phase, a random sequence of noisy seismic data was applied, too. All earthquake events can be recognized with a high confidence and prediction accuracy (suing complete set). Some tests were made with incomplete training sets (last three rows in Figure *10.13*) to find similar events.

The incremental *DI²NN* learner algorithm was about 200% faster than the non-incremental *DINN* algorithm with the same training data sets (accumulated computation time observed in the entire network with all participating learners). The accuracy of the incremental learner is comparable to the *DINN* learner. The confidence of the election result can be slightly improved if weighted votes are processes, i.e., the local accumulated sensor data is used as a weight, dominating the election by nodes with a high stimulus. But the *DI²NN* does not profit from vote weighting.

The transition from learning to prediction is seamless and is based on the node/learner experience (learned events). Furthermore, after an event is

elected by the majority decision, this result can be back propagated to the learner adding the new data set as a new training set and performing incremental learning to improve further prediction accuracy. A typical learning and ROI exploration run in the entire network requires about 3-5MB total communication cost if code compression is enabled, which is a reasonable low overhead (with a peak value about 500-1000 mobile explorer agents operating in the network). Vote distribution produces only a low additional communication overhead (less than 1MB in the entire network). The usage of I^2NN lowers the entire communication costs about 30% compared with the *INN* approach (due to the data base).

10.7 Further Reading

1. P. Attewell and D. B. Monaghan, *Data mining for the social sciences: an introduction*, University of California Press, 2015, ISBN 9780520280977
2. T. Mueller, A. G. Kusne, and R. Ramprasad, *Machine learning in materials science: Recent progress and emerging applications*, Reviews in Computational Chemistry, Volume 29, First Edition., 2016.
3. L. Rokach and O. Maimon, *Data Mining with Decision Trees - Theory and Applications*, World Scientific Publishing, 2015, ISBN 9789814590075
4. J. Bell, *Machine Learning - Hands-On for Developers and Technical Professionals*, John Wiley & Sons, Ltd, 2015, ISBN 9781118-889060
5. T. Dietterich, C. Bishop, M. J. Heckermann, and M. Kearns, *Introduction to Machine Learning*, Second Edition, MIT Press Cambridge, 2009, ISBN 9780262012430
6. T. M. Mitchel, *Machine Learning*, McGraw Hill, 1997, ISBN 0070428077
7. C. R. Farrar and K. Worden, *Structural Health Monitoring: A Machine Learning Perspective*, Wiley-Interscience, 2013, ISBN 9781119994336.