Incrementally Updating a Hybrid Rule Base Based on Empirical Data

Jim Prentzas[(1),(2)] and Ioannis Hatzilygeroudis[(1)]

[(1)]University of Patras, School of Engineering
Department of Computer Engineering & Informatics
26500 Patras, Greece
&
[(2)]Technological Educational Institute of Lamia
Department of Informatics and Computer Technology
35100 Lamia, Greece

ihatz@ceid.upatras.gr, dprentzas@teilam.gr

Abstract

Neurules are a kind of hybrid rules that combine a symbolic (production rules) and a connectionist (adaline unit) representation. One way that the neurules can be produced is from training examples/patterns, extracted from empirical data. However, in certain application fields not all of the training examples are available a priori. A number of them become available over time. In those cases, updating a neurule base is necessary. In this paper, methods for updating a hybrid rule base, consisting of neurules, to reflect the availability of new training examples are presented. They can be considered as a type of incremental learning methods that retain the entire induced hypothesis and all past training examples. The methods are efficient, since they require the least possible retraining effort and the number of the produced neurules is kept as small as possible. Experimental results that prove the above argument are presented.

1. Introduction

There have been many efforts at combining (or integrating) the symbolic and the connectionist approaches for problem solving and/or machine learning (McGarry, Wertmer and MacIntyre 1999; Wermter and Sun 2000; d'Avila Garcez, Broda and Gabbay 2002). Especially, there have been a number of efforts at combining symbolic rules and neural networks (Fu and Fu 1990; Towell and Shavlik 1994; Hall, Sanou and Romaniuk 1996). In addition, connectionist expert systems (Gallant 1993; Quah, Tan, Krishnamurthy and Srinivasan 1996; Ghalwash 1998) have been considered as a type of integrated systems that represent relationships between concepts, considered as nodes of a neural network. All the above approaches give pre-eminence to connectionism, that is they incorporate or map symbolic rule concepts and/or processes into a neural network, which constitutes their knowledge base. The strong point of those approaches is that knowledge elicitation from experts is reduced to a minimum. A weak point is that their knowledge base lacks the naturalness and modularity of symbolic rules; it is incomprehensible. Therefore, often explanations are provided in the form of if-then rules by rule extraction methods (Andrews, Diederich and Tickle 1995, d' Avila Garcez et al 2001, Palade et al 2001).

Neurules (Hatzilygeroudis and Prentzas 2000) integrate symbolic rules and connectionism too, but in a different way. They give pre-eminence to the symbolic component: neurocomputing is used within the symbolic framework to improve the

J. Prentzas, I. Hatzilygeroudis, "Incrementally updating a hybrid rule base based on empirical data", Expert Systems, 24(4), pp. 212-231, 2007.

performance of symbolic rules. The constructed knowledge base retains the modularity of production rules, since it consists of autonomous units (neurules), and also retains their naturalness in a great degree, since neurules look much like symbolic rules. Also, the inference mechanism is a tightly integrated process, which results in more efficient inferences than those of symbolic rules and explanations in the form of if-then rules can be produced (Hatzilygeroudis and Prentzas 2001b).

One way that a neurule base, called the *target knowledge*, can be produced is from training examples (Hatzilygeroudis and Prentzas 2001a), called the *source knowledge*. However, in certain application fields (e.g. user modeling/profiling, intelligent agents, intelligent user interfaces and robotics) not all of the training examples are available a priori. A number of them become available over time. This happens either because the environment changes with time or because the rate at which examples become available may be too slow (Giraud-Carrier 2000). So, this raises the issue of incremental update of (or learning with) a neurule base. Therefore, methods should be developed dealing with this problem. Such methods may or may not require re-examination of all or part of the source knowledge (Maloof and Michalski 2004). On the other hand, those methods must be effective as far as the retraining effort and the size of the target knowledge are concerned.

In this paper, we present methods for efficient maintenance of the target knowledge of a neurule-based system, due to changes to its source knowledge. This paper is an extension of (Prentzas and Hatzilygeroudis 2002) and can be considered as complementary to (Prentzas and Hatzilygeroudis 2005).

The paper is organized as follows. Section 2 presents neurules and the neurule base construction process. The update algorithms are introduced in Section 3. Section 4 gives some examples and Section 5 presents experimental results. Section 6 deals with related work. Finally, Section 7 concludes the paper.

2. Neurules

2.1 Syntax and Semantics

Neurules are a kind of hybrid rules. The form of a neurule is depicted in Fig.1a. Each condition C_i is assigned a number sf_i, called its *significance factor*. Moreover, each rule itself is assigned a number sf_0, called its *bias factor*. Internally, each neurule is considered as an adaline unit (Fig.1b). The *inputs* C_i (i=1,...,n) of the unit are the conditions of the rule. The weights of the unit are the significance factors of the neurule and its bias is the bias factor of the neurule. Each input takes a value from the following set of discrete values: [1 (true), -1 (false), 0 (unknown)]. The *output* D, which represents the conclusion (decision) of the rule, is calculated via the standard formulas (see e.g. (Gallant 1993)):

$$D = f(\mathbf{a}), \quad \mathbf{a} = sf_0 + \sum_{i=1}^{n} sf_i \; C_i$$

$$f(\mathbf{a}) = \begin{cases} 1 & \text{if a} \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

where **a** is the *activation value* and *f(x)* the *activation function*, which is a threshold function. Hence, the output can take one of two values ('-1', '1') representing failure and success of the rule respectively. The significance factor of a condition represents the significance (weight) of the condition in drawing the conclusion.

The general syntax of a condition C_i and the conclusion D is:

<condition>::= <variable> <l-predicate> <value>
<conclusion>::= <variable> <r-predicate> <value>
where <variable> denotes a *variable*, that is a symbol representing a concept in the domain, e.g. 'sex', 'pain' etc, in a medical domain. <l-predicate> denotes a symbolic or a numeric predicate. The symbolic predicates are {is, isnot}, whereas the numeric predicates are {<, >, =}. <r-predicate> can only be a symbolic predicate. <value> denotes a value. It can be a symbol or a number. An example neurule is depicted in Table 1.

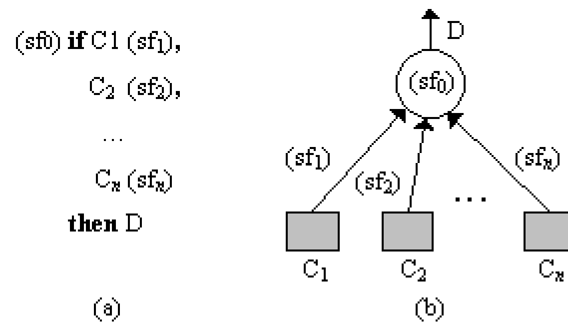

Figure 1. (a) Form of a neurule (b) a neurule as an adaline unit

Table 1. An example neurule

(-20.4) **if** venous-concentration is slight-incr (8.7),
arterial-concentration is mod-incr (8.4),
scan-concentration is normal (8.1),
capillary-concentration is slight-incr (8.1),
blood-concentration is normal (8.0),
blood-concentration is highly-incr (4.6)
venous-concentration is normal (1.5),
then disease-type is early-inflammation

2.2 Constructing a Neurule-Base

2.2.1 Neurules production algorithm

One way of constructing a neurule base (NRB) is from empirical data (i.e. training examples/patterns). What is required is the *dependency information* (DI), concerning the domain variables (concepts), and a set of empirical data, which we call the *source set* (SS). DI and SS constitute the *source knowledge*. DI indicates which variables the intermediate, if any, and output variables depend on. Production of an NRB is achieved by applying the *neurules production algorithm* (NPA). Application of NPA to source knowledge results in the construction of the correspnding NRB (Fig. 2).

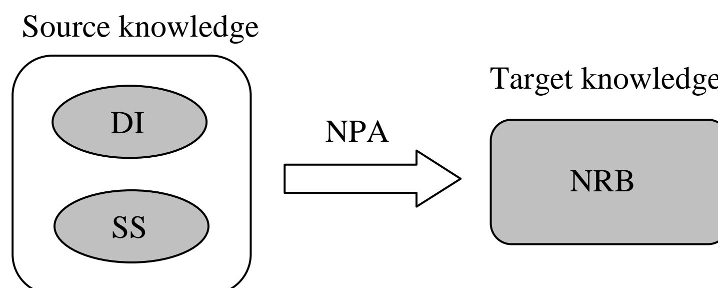

Figure 2. Neurule base production

J. Prentzas, I. Hatzilygeroudis, "Incrementally updating a hybrid rule base based on empirical data", Expert Systems, 24(4), pp. 212-231, 2007.

NPA tries to produce one neurule for each output variable-value pair. However, due to possible non-linearity of the SS, this is not usually feasible. So, more than one neurule having the same conclusion are produced. The basic steps of NPA are as follows:

1. Construct *initial neurules*, based on the DI.
2. Extract an *initial training set* for each initial neurule from the SS.
3. Train each initial neurule individually and produce one or more neurules.

The *initial neurules* are constructed, based on the DI. One initial neurule is constructed for each conclusion corresponding to an intermediate or output variable. The conditions of each initial neurule include the variables that contribute in drawing the corresponding conclusion. Then, for each initial neurule its corresponding *initial training set* is extracted from the SS. A *training example/pattern* has the form [v_1 v_2 … v_n *d*], where d is the desired value of a variable related to a partial or final conclusion and v_i, i= 1, …,n are the values of the variables it depends on, called *component values*. We distinguish between *success examples* and *failure examples* in a training set. Success examples are those having '1' as their d value, whereas failure examples those having a '-1'. Furthermore, the *closeness* between two examples is defined as the number of their common component values. So, a *least closeness pair* (LCP) consists of two success examples that have the least closeness between them (for more details see Hatzilygeroudis and Prentzas 2001a).

Each initial neurule is individually trained via the Least Mean Square (LMS) algorithm using its own training set. Training is not always successful, that is a set of significance and bias factors cannot always be found that correctly classify all of the training examples. Training is not successful in case that the training examples of the initial training set are inseparable. When the algorithm succeeds, that is values for the bias and significance factors are calculated that classify all training examples, a neurule is produced. When it fails, due to inseparability of the training examples, a splitting process is followed. More specifically, the initial training set of the neurule is split into two subsets and two copies of the initial neurule are trained, each using one of the training subsets. Splitting a training set is based on a LCP. That is, each subset comprises one of the members of a LCP, the success examples closer to it and all the failure examples of the initial training set. This stems from the intuition that existence of quite different examples causes non-separability (or non-linearity). If training of either neurule copy fails, its subset is further split into two other subsets and so on, until there is no failure. In this way, more than one neurule are produced, having the same conditions with different bias and significance factors and the same conclusion, called *sibling neurules*.

So, step 3 of NPA is analysed (for each initial neurule) as follows:

3.1 Train the initial neurule using the specified initial training set. If training is successful, produce corresponding neurule.
3.2 If training fails, find an LCP and produce two subsets of the initial training set, each including as initial element one of the LCP examples respectively. In each subset also include the success examples closer to the corresponding LCP example and all failure examples of the initial training set.
3.3 For each subset apply step 3.1 recursively, until training is successful and produce corresponding neurule.

2.2.2 The splitting tree

For reasons that will become clear in the following parts of the paper, we present here the notion of a *splitting tree*. For each initial training set, its splitting process (if there is one) is stored as a tree, which is called its splitting tree. The root of the tree corresponds to the initial training set. The intermediate nodes and leaves correspond to the subsequent subsets into which the initial training set was split in. An intermediate node denotes a subset from a split, due to training failure, whereas a leaf denotes a subset that was successfully trained and produced a neurule. The members of the LCP that guided each split are attached to the corresponding branches of the tree. It can be easily seen that the training (sub)set of the root or an intermediate node is a superset of the training subsets related to its descendant nodes. Furthermore, the nearer one gets to the leaves, the greater the mean closeness between the training examples of the corresponding training subsets. Tree information is assigned to each initial training set that had to split.

2.2.3 An example

To demonstrate application of NPA, we use as an example the training set presented in Table 2. As it is clear, the majority of the examples in the training set are failure examples, whereas success examples, which are shown in bold, are a minority. Actually Table 2, for simplicity reasons, shows only a subset of the failure examples; the real training set includes 448 examples in total.

The training set has been extracted from empirical data concerning five input (domain) variables (arterial-concentration, blood-concentration, scan-concentration, capillary-concentration, venous-concentration) and an output variable (disease) that depends on the five domain variables. Given that each input variable can take more than one discrete value, each initial neurule has thirteen conditions (C1-C13), presented in Table 3. D corresponds to the conclusion.

Table 2. An example training set

C1	*C2*	*C3*	*C4*	*C5*	*C6*	*C7*	*C8*	*C9*	*C10*	*C11*	*C12*	*C13*	*D*
-1	-1	1	-1	-1	1	1	1	-1	1	-1	-1	-1	-1
-1	**-1**	**1**	**-1**	**-1**	**1**	**1**	**1**	**1**	**-1**	**-1**	**-1**	**-1**	**1**
-1	-1	1	1	-1	1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	1	1	-1	1	-1	-1	-1	-1	-1	-1	1	-1
-1	-1	1	1	-1	1	-1	-1	-1	-1	-1	1	-1	-1
-1	**-1**	**1**	**1**	**-1**	**1**	**-1**	**-1**	**-1**	**-1**	**-1**	**1**	**1**	**1**
-1	-1	1	1	-1	1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	1	1	-1	1	-1	-1	-1	1	-1	1	-1	-1
-1	-1	1	1	-1	1	-1	-1	1	-1	-1	1	-1	-1
-1	**-1**	**1**	**1**	**-1**	**1**	**-1**	**-1**	**1**	**-1**	**1**	**-1**	**-1**	**1**
-1	1	1	-1	-1	-1	-1	1	-1	-1	-1	-1	1	-1
-1	1	1	-1	-1	1	-1	1	-1	-1	-1	-1	-1	-1
-1	1	1	-1	-1	1	-1	1	-1	-1	-1	-1	1	-1
-1	**1**	**1**	**-1**	**-1**	**1**	**-1**	**1**	**-1**	**1**	**-1**	**-1**	**-1**	**1**
1	1	1	1	-1	-1	-1	-1	-1	-1	-1	-1	1	-1
1	1	1	1	-1	-1	-1	-1	-1	1	-1	-1	-1	-1
1	1	1	1	-1	-1	-1	-1	1	-1	-1	-1	-1	-1
1	**1**	**1**	**1**	**1**	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**	**1**

For presentation reasons, names (P1-P5) are assigned to the five success examples/patterns (of Table 2), as presented in Table 4. Also, let F be the set of failure examples in the training set.

Table 3. Example conditions and conclusion

Symbol	Description
C1	arterial-concentration is slight
C2	blood-concentration is normal
C3	scan-concentration is normal
C4	capillary-concentration is moderate
C5	venous-concentration is high
C6	arterial-concentration is moderate
C7	blood-concentration is high
C8	capillary-concentration is slight
C9	venous-concentration is slight
C10	venous-concentration is normal
C11	blood-concentration is moderate
C12	blood-concentration is slight
C13	venous-concentration is moderate
D	disease is early-inflammation

Table 4. Success examples

Symbol	Description
P1	[-1, -1, 1, -1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1]
P2	[-1, -1, 1, 1, -1, 1, -1, -1, -1, -1, -1, 1, 1, 1]
P3	[-1, -1, 1, 1, -1, 1, -1, -1, 1, -1, 1, -1, -1, 1]
P4	[-1, 1, 1, -1, -1, 1, -1, 1, -1, 1, -1, -1, -1, 1]
P5	[1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, -1, -1, 1]

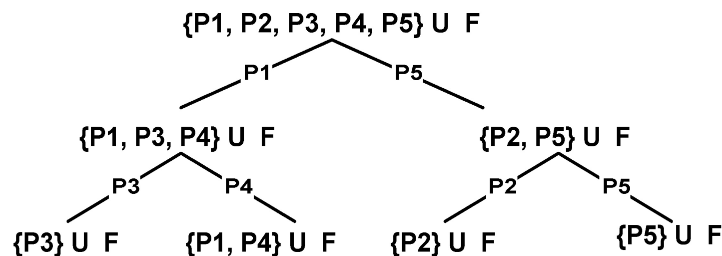

Figure 3. The splitting tree for the training set of Table 1

From the above training set (the complete one), after application of the NPA, four neurules are finally produced (two of them are depicted in Table 5). Figure 3 illustrates the corresponding splitting tree, which represents the splits that have taken place during the process.

Due to inseparability, the initial training set {P1, P2, P3, P4, P5} $\cup$ F is split in two subsets: {P1, P3, P4} $\cup$ F and {P2, P5} $\cup$ F with as LCP: (P1, P5). Subset {P1, P3, P4} $\cup$ F is subsequently split into subsets {P3} $\cup$ F and {P1, P4} $\cup$ F. Subset {P3} $\cup$ F produces a neurule (NR1, Table 5). Subset {P1, P4} $\cup$ F produces another neurule (NR2, Table 5). Similarly, from subset {P2, P5} $\cup$ F two other neurules are produced (corresponding to its two leaves).

Table 5. Two of the neurules produced from the example training set

NR1	NR2
(-13.5) **if** venous-conc is slight (12.4), blood-conc is moderate (11.6), arterial-conc is moderate (8.8), scan-conc is normal (8.4), capillary-conc is moderate (8.4), blood-conc is slight (8.3), venous-conc is moderate (8.2), venous-conc is normal (8.0), arterial-conc is slight (-5.7), capillary-conc is slight (4.5), blood-conc is normal (4.4), blood-conc is high (1.6), venous-conc is high (1.2) **then** disease is early-inflammation	(-14.6) **if** blood-conc is normal (14.0), venous-conc is slight (13.5), arterial-conc is moderate (10.6), capillary-conc is slight (10.4), scan-conc is normal (10.1), venous-conc is normal (9.9), blood-conc is high (9.9), venous-conc is moderate (6.5), blood-conc is moderate (6.3), blood-conc is slight (3.2), venous-conc is high (-1.0), capillary-conc is moderate (-0.5), arterial-conc is slight (-0.4) **then** disease is early-inflammation

3. Incremental Update

3.1 Architecture and Requirements

The existence of application domains in which a number of training examples become available over time imposed the necessity to develop update methods for neurules. It should be mentioned that the update methods for neurules described here require the storage of all past examples. However, depending on the update method, as will be analysed in the following parts of the section, not all past examples need to be processed in (re)training.

In Figure 4, the architecture of the maintenance system of a neurule-base is illustrated. IUM is the *incremental update mechanism*, which implements the update methods/algorithms, introduced in the next subsections. NPM is the *neurule production mechanism*, which basically implements NPA. Finally, P is a pattern, which represents a new empirical example.

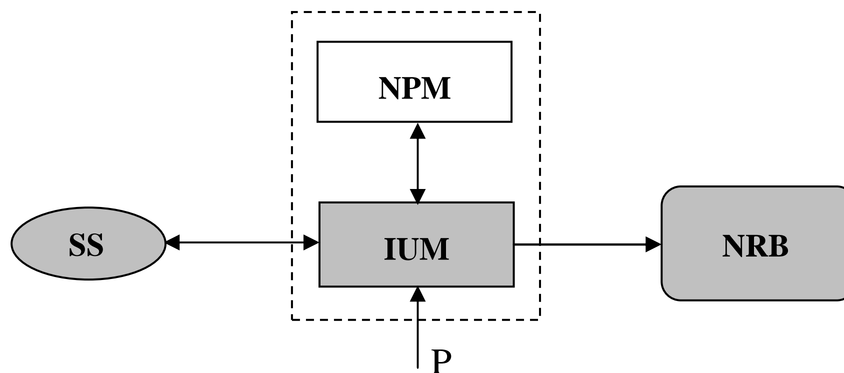

Figure 4. Architecture of a neurule-base incremental update system

Availability of new empirical data in the SS means insertion of a new example (either success or failure) into all the initial training sets that are affected. Given the modularity of a neurule-base, when a new example is available and is not satisfied by the existing neurules, it does not affect the whole base, but one or more sets of sibling

J. Prentzas, I. Hatzilygeroudis, "Incrementally updating a hybrid rule base based on empirical data", Expert Systems, 24(4), pp. 212-231, 2007.

rules. So, the basic problem is how to update a set of sibling neurules, due to the availability of an extra training example/pattern, which should be taken into account alongside their initial training set. The basic steps when a new example/pattern P becomes available are the following:

1. Check if the existing sibling neurules satisfy P.
2. If P is satisfied, store it along with the existing examples.
3. Otherwise, perform the update process.

The updating process, in order to be efficient, should have the following two features:

(a) results in (re)training the least possible subset of corresponding sibling neurules and
(b) keep the number of corresponding sibling neurules as small as possible.

The first feature guarantees that the computational cost of the updates will be as low as possible. The second one assures that the efficiency of the inferences will not significantly decrease. The number of neurules in the NRB affects the number of computations required for the evaluation of the rule conditions (Hatzilygeroudis and Prentzas 2000). Furthermore, it is desirable to keep the number of neurules with the same conclusion as small as possible, so that the size of the conflict set during inference will be as small as possible too.

3.2 Update methods

At the insertion of a new example in an initial training set, two cases can be distinguished: (a) there is no splitting tree associated with the training set, (b) there is a splitting tree. Case (a) is a simple one. The fact that there is no splitting tree means that the initial training set was not split, hence only one neurule was produced. To handle this case, the existing neurule is removed from the NRB and (re)training with the updated initial training set is performed. If training is successful, one new (updated) neurule is produced. If training fails, two new neurules are produced. Case (a) is handled in the same way for both, the insertion of a success and the insertion of a failure example.

Case (b) is a difficult one. The existence of a splitting tree means that there was at least one splitting of the initial training set and two or more sibling neurules were produced. There can be various approaches to handle this case, which are presented in the next subsections.

3.2.1 Insertion of a Success Example

At inserting a success example/pattern P not satisfied by the existing neurules in an initial training set, there can be three approaches to handle case (b).

(i) Leave all corresponding sibling neurules intact and insert into the NRB an extra (sibling) neurule produced from a training set containing the new example and the failure examples of the initial training set. This method, called the *simple* (SI) *method*, is the computationally most efficient, but definitely increases the number of neurules in the NRB, negatively affecting the inference process. SI method does not take into account the information stored in the splitting tree.

(ii) Corresponding sibling neurules are removed from the NRB, the new example is inserted into the initial training set and retraining is performed to produce the new

(sibling) neurules. This approach is actually based on retraining the whole set of the sibling neurules, therefore it is called the *total retraining* (TR) *method*. TR method is inefficient, especially when more than two neurules are produced from the initial training set. The reason is that it discards the information stored in the splitting tree, thus performing extra training and splitting.

(iii) The third approach, call the *splitting tree* (ST) *method*, exploits the information stored in the splitting tree. It focuses on the training subset containing the success examples that are closer to P. To this end, the splitting tree is traversed starting from the root and ending at a leaf or an intermediate node. Traversing is based on the closeness between the new success example P and the least closeness pairs attached to the edges of the splitting tree. More formally, the corresponding algorithm is as follows:

1. Set the root of the splitting tree as the current node.

2. If the current node is not a leaf, check whether the training (sub)set corresponding to the node contains an example P' whose closeness to P is less than the least closeness of the (sub)set.

2.1 If there is no such example, insert P into the training (sub)set of the node and execute this step recursively for the child of the node at the end of the branch labeled by the member of the LCP, which is closer to P.

2.2 If there is such an example P', do the following:

2.2.1 Remove from the splitting tree all the nodes descending from the current node and from the NRB all (sibling) neurules corresponding to the leaves descending from current node. Store thet removed nodes and neurules into a temporary buffer.

2.2.2 Insert P into the corresponding training (sub)set and split it in two subsets with as LCP: (P, P').

2.2.3 Perform (re)training based on the two training subsets, produce the corresponding neurules (reusing parts of the initial splitting tree to avoid unnecessary training or splitting), insert the produced neurules into the NRB and update the splitting tree. Moreover, clear the temporary buffer.

3. If the current node is a leaf, remove the corresponding neurule, insert P into its training set and perform (re)training.

3.1 If (re)training fails, split the training set, produce the corresponding neurules (reusing parts of the initial splitting tree to avoid unnecessary training or splitting), insert them into the NRB and update the splitting tree.

3.2 If training is successful do:

3.2.1 If the sibling node of the leaf is also a leaf and introduction of P into their parent's training (sub)set increases the mean closeness between its examples, and the parent's new training (sub)set can be successfully classified do:

3.2.1.1 Remove the neurule corresponding to the sibling node from the NRB.

3.2.1.2 Insert the neurule produced from the parent's node new training (sub)set into the NRB.

3.2.1.3 Update the splitting tree.

3.2.2 Else do:

3.2.2.1 Insert the neurule produced from the leaf's new training (sub)set into the NRB.

J. Prentzas, I. Hatzilygeroudis, "Incrementally updating a hybrid rule base based on empirical data", Expert Systems, 24(4), pp. 212-231, 2007.

2.2.2.2 Update the splitting tree.

The ST method results in changes to the least possible subset of the corresponding sibling neurules set and therefore requires less (re)training effort than the TR method. Furthermore, the ST method produces less sibling neurules than SI. The SI method inserts a neurule into the NRB when a new success example is inserted into the initial training set. On the contrary, if the insertion of a success example is handled according to the ST method, the number of the produced neurules may not increase. More specifically, if traversing reaches a leaf and training of the subsequent subset is successful, the number of produced neurules remains the same or may even decrease (step 3.2). It should be mentioned that in step 3.2, the mean closeness of the parent training (sub)set is checked because, if it is increased, it is likely that the corresponding new training (sub)set can be successfully trained.

As can be easily seen from the description of ST method, most of the changes are required when traversing stops at an intermediate node. In that case, the new training (sub)set corresponding to the new intermediate node has to be split again, due to the fact that the least closeness has been decreased compared to the training (sub)set prior to the introduction of the new example. Splitting is based on the new least closeness.

If TR or SI method is used to perform the updates, the insertion of newly available success examples that are satisfied by the existing neurules (and for which no retraining is required) is handled more conveniently. Such examples are merely retained alongside the existing examples. If the ST method is used to perform the updates, the traversal of the splitting tree and the storage of such examples in the corresponding nodes is required.

(iv) The ST method can be modified to deal better with situations of simultaneous multiple pattern insertions in the same initial training set. Such a method, called the *simultaneous splitting tree* (SST) *method*, is advantageous, compared to single ST method, in cases in which multiple inserted patterns end up at the same nodes after splitting tree traversal. Because training is the computationally most expensive process, it would be wiser, in such cases, to train the corresponding subsets only once (after inserting all corresponding new patterns) instead of multiple times. It is presumed that the number of patterns simultaneously inserted into the same initial training set should be remarkably smaller than the number of past examples. The basic ideas underlying the SST method for multiple success pattern insertions are:

(a) New success examples should traverse the splitting tree one by one.
(b) (Re)training based on subsets should be performed after the traversal of the splitting tree by all the new examples.

So, postponing (re)training till all (simultaneously arrived) training examples have traversed the splitting tree is the basic characteristic of the SST method. More specifically, the SST algorithm is as follows:

1. *For each new success example (one by one) traverse the splitting tree. If traversing of an example reaches a leaf, do not perform (re)training but instead mark the corresponding node. If traversing of an example stops at an intermediate node, merely mark the node without doing anything else. Traversing of a new example stops at an intermediate node if it has been already marked.*

2. *After all new examples have traversed the splitting tree, check all marked nodes. To check which nodes have been marked, the preorder traversal of the tree starting from the root is employed (this type of traversal traverses recursively the tree nodes in the order parent-node, left-child, right-child). For the marked nodes*

(either leaf or intermediate nodes) found, actions similar to the single ST method are performed:

2.1. *If a marked leaf node is found during checking traversal, the corresponding neurule is removed from the NRB and the new training subset is processed. If training fails, act as in step 3.1 of the single ST method. If training is successful, act as in step 3.2 of the single ST method (if step 3.2.1 is executed, the mean closeness of parent node's training subset, prior and after the insertion of all its new patterns, is checked). At the end, marking of the node is removed.*

2.2. *If a marked intermediate node is found during checking, execute step 2.2 of the single ST method, split the node's subset based on the new LCP and execute step 2.4 of the single ST method. At the end, marking of the node is removed.*

3.2.2 Insertion of a Failure Example

The insertion of a failure example not satisfied by the existing neurules into an initial training set affects all the sibling neurules produced from this set. At inserting a failure example/pattern in an initial training set, there can be two approaches to handle case (b).

(i) Corresponding sibling neurules are removed from the NRB, the new example is inserted into the initial training set and retraining is performed to produce the new neurules. Again, this approach is actually based on retraining the whole set of the sibling neurules, is inefficient and is called the *total retraining* (TR) *method*. This is due to the fact that it discards the information stored in the splitting tree, thus performing extra training and splitting.

(ii) The second approach, called the *splitting tree* (ST) *method*, exploits the information contained in the splitting tree. It removes corresponding sibling neurules from the NRB and inserts the example into the training subsets of all nodes of the splitting tree. It performs training of the subsets corresponding to leaves of the splitting tree whose corresponding neurules misclassify the new failure example. The corresponding existing neurules are removed from the NRB, whereas the newly created ones are inserted. It is obvious that the ST approach usually requires more training effort than the corresponding ST approach, related to the insertion of success examples. However, this ST approach requires less training effort than that.

If the TR approach is used to perform the updates, the newly available failure examples that are satisfied by existing neurules are merely retained alongside the existing examples. ST approach requires their storage in all nodes of the splitting tree.

4. Some Examples

Consider the initial training set presented in section 2.2. Suppose that the success example P6 = [-1, 1, 1, 1, -1, 1, -1, -1, 1, -1, -1, -1, -1, 1] is to be inserted into the initial training set. Given that more than one sibling neurule were produced from the initial training set, it is a (b) case.

Following the ST approach, traversing ends at the leaf corresponding to subset {P3} $\cup$ F (see Figs 3 and 5). Training based on the new training subset {P3, P6} $\cup$ F is successful. The sibling node is also a leaf and the insertion of P6 into their parent's training subset increases its mean closeness. Training of subset {P1, P3, P4, P6} $\cup$ F is tried, which is unsuccessful, and the process stops. The splitting tree takes the form

in Fig. 6. So, the total number of neurules in the NRB, after the insertion of P6, remains four (corresponding to the leaves of the splitting tree in Fig. 6).

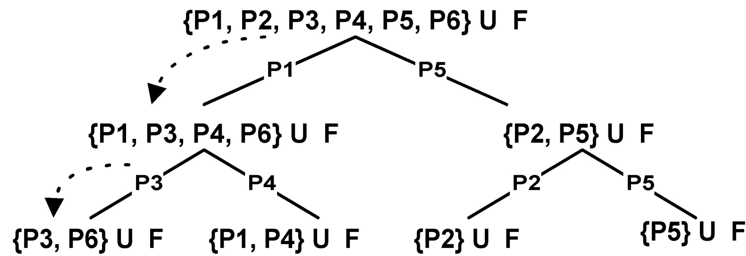

Figure 5. Traversal of the splitting tree for the insertion of example P6

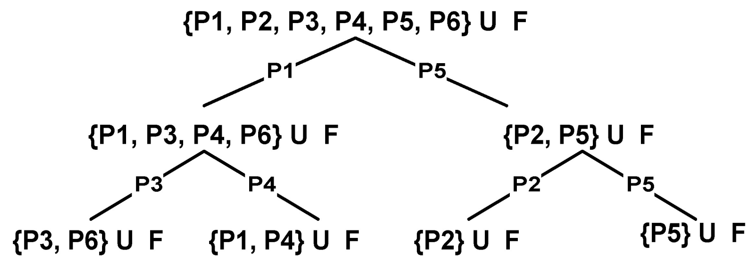

Figure 6. The splitting tree after insertion of example P6

Notice that, the ST approach finally ends at subset {P3} ∪ F (Fig. 3), which includes the success example closest to P6. So, only one of the corresponding sibling neurules requires (re)training. The rest of them remain intact. The TR approach produces the same number of neurules, requiring though unnecessary training and splitting. The SI approach inserts the neurule produced from subset {P6} ∪ F into the NRB. So, the SI method produces more neurules than ST method.

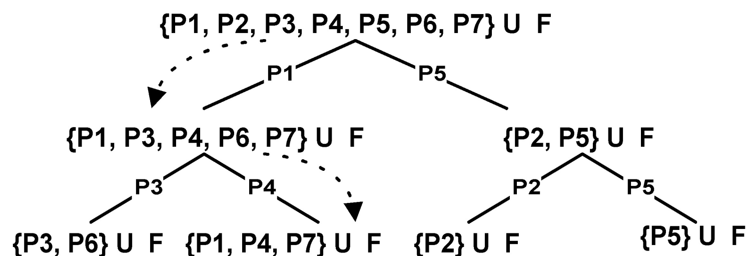

Figure 7. Traversal of the splitting tree for the insertion of example P7

Suppose that (after the insertion of P6) the success example P7 = [-1, 1, 1, -1, -1, 1, -1, 1, 1, -1, -1, -1, -1, 1] is to be inserted into the training set {P1, P2, P3, P4, P5, P6} ∪ F. Following the ST approach, traversing ends at the leaf related to subset {P1, P4} ∪ F (see Figs 6 and 7). Training based on the new training subset {P1, P4, P7} ∪ F is successful. The sibling node is also a leaf and the insertion of P7 into their father's training subset increases its mean closeness. Training of subset {P1, P3, P4, P6, P7} ∪ F is tried which is unsuccessful and the process stops. The splitting tree takes the form shown in Fig. 8. The total number of neurules in the neurule-base, after inserting P7, remains again four (corresponding to the leaves of the splitting tree in Fig. 8).

J. Prentzas, I. Hatzilygeroudis, "Incrementally updating a hybrid rule base based on empirical data", Expert Systems, 24(4), pp. 212-231, 2007.

Once again only one of the sibling neurules requires (re)training. The rest remain intact.

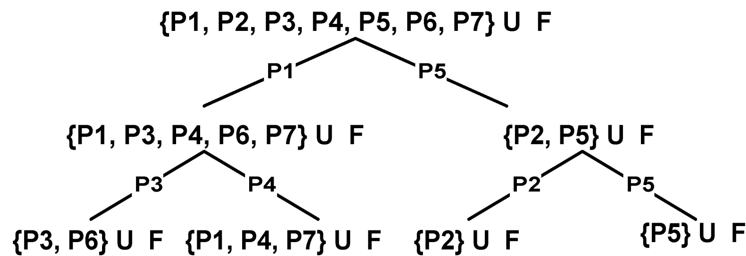

Figure 8. The splitting tree after insertion of example P7

The TR approach produces the same neurules, requiring though unnecessary training and splitting. The ST approach inserts the neurule produced from subset {P7} ∪ F into the NRB. So, the SI approach, in order to update the NRB, due to insertion of both P6 and P7, increases the number of neurules from four to six, whereas ST method results in no change to that number; they remain four.

5. Experiments and Discussion

5.1 Introductory Issues

The fact that our methods refer to neurules, a special hybrid rule structure with no similar counterparts, make them experimentally not comparable to other existing methods. Therefore, our experiments refer to possible alternative update methods for neurule bases, as presented in previous sections.

So, in this section, we present experimental results comparing the different approaches for handling case (b), which is the most interesting case of an example insertion. For this purpose, we used seven datasets of ours produced from a medical domain (Hatzilygeroudis, Vassilakos and Tsakalidis 1997): the *arthritis* dataset, containing 144 examples/patterns of 9 component values, the *primary_malignant* dataset, containing 120 examples/patterns of 10 component values, the *early-inflammation* dataset containing 540 examples/patterns of 13 component values, the *soft-tissue-inflammation* dataset containing 480 examples/patterns of 14 component values, the *soft-tissue-early-bone-inflammation* dataset containing 576 examples/patterns of 13 component values, the *soft-tissue-bone-inflammation* dataset containing 144 examples/patterns of 9 component values and the *early-soft-tissue-inflammation* dataset containing 360 examples/patterns of 13 component values.

Table 6 shows the conclusions related to each dataset of the medical domain. Table 7 contains the general characteristics of the initial training sets. For each initial training set, it shows the total number of the training patterns, the number of conditions of the produced neurules (corresponding to the component values of the patterns) and the number of neurules produced from the initial training set. It should be noted that all the neurules produced from the seven datasets form a NRB producing inferences in the specific medical domain. Section 5.2 presents experimental results for comparing the update methods dealing with the insertion of a success example, whereas section 5.3 presents experimental results for comparing the update methods dealing with the insertion of a failure example.

Table 6. Datasets and corresponding conclusions

Dataset	Conclusion
D1	disease-type is arthritis
D2	disease-type is primary-malignant
D3	disease-type is early-inflammation
D4	disease-type is soft-tissue-inflammation
D5	disease-type is soft-tissue-early-bone-inflammation
D6	disease-type is soft-tissue-bone-inflammation
D7	disease-type is early-soft-tissue-inflammation

Table 7. General characteristics of the initial training sets

Dataset	Total Patterns	Number of Conditions	Number of Produced Neurules
D1	144	9	3
D2	120	10	2
D3	540	13	5
D4	480	14	6
D5	576	13	3
D6	144	9	2
D7	360	13	4

The comparison between the update methods is based on two measures, which stem from the requirements set in Section 3. The first is the total number of produced neurules (directly stemming from the first requirement). The second is the number of training (sub)sets that had to be (re)trained, because of the new empirical source knowledge, (stemming from the second requirement). Indeed, the number of required (re)trainings is a measure of the required neurule (re)production effort, because in the algorithms presented in section 3, training an initial neurule is the far computationally more expensive process, thus has the most significant impact on the required runtime. Although the number of required trainings is an adequate measure, to increase the certainty of our conclusions, we also made runtime measurements in our experiments. It is a fact that not all trainings are equally expensive, because they depend on the size of the training (sub)sets. Training (sub)sets higher up in the splitting tree are bigger in size than those lower down, hence training based on the former is more expensive than that based on the latter. This cannot be captured by using as a measure only the number of trainings.

Furthermore, our experiments correspond to a number of possible scenarios when updating empirical source knowledge:

SC1. Single Pattern Insertion.
SC2. Multiple Pattern Insertions-Single Splitting Tree.
SC3. Multiple Pattern Insertions-Multiple Splitting Trees.
SC4. Simultaneous Multiple Pattern Insertions-Single Splitting Tree.
SC5. Simultaneous Multiple Pattern Insertions-Multiple Splitting Trees.

In SC1, a single pattern/example is inserted into SS. In this case only one splitting tree (or initial training set) is affected. In SC2, a number of patterns corresponding to the same conclusion are inserted in the source knowledge one by one. One by one means that each update (due to the pattern insertion) takes place at a different time. Again, only one splitting tree is affected (because the patterns correspond to the same conclusion). In SC3, a number of patterns that do not all correspond to the same conclusion are inserted into SS, one by one. In this scenario, more than one splitting

tree is affected (due to the different conclusions). SC4 is the same as SC2, but all pattern insertions are simultaneously made (i.e. at the same time). The same holds for SC5 and SC3. It should be noted that scenario SC4 is an unlikely one. The usual scenario, also taken into consideration in other incremental approaches (e.g. Utgoff 1989), is to have single pattern insertions. However, we will consider all scenarios in order to record the behavior of the update methods (and especially ST method) in all cases. One can suppose though that the number of patterns simultaneously inserted into the same initial training set should be quite smaller than that of past examples.

To do our experiments, we implemented our algorithms in ANSI C using MS Visual Studio 6.0. The experiments were run on an Intel Pentium 4 (3.0 GHz) CPU PC with 512 MB RAM.

5.2 Results for Success Example Insertion

Tables 8 and 9 present experimental results for the update methods dealing with insertion of success examples into the initial training sets of Table 7. To produce the results, some success examples were removed from those sets and inserted afterwards one by one.

Table 8. Experimental results-Example insertions (SC1, SC3 scenarios)

Dataset	**Initial Examples**	**Initial Neurules**	**Training Subsets**	**Simple (SI) Method**	**Total Retraining (TR) Method**		**Splitting Tree (ST) Method**	
				Neurules	*Neurules*	*Trained Subsets*	*Neurules*	*Trained Subsets*
D1	143 (1)	3	5	4	3	5	3	2
D2	119 (1)	2	3	3	2	3	2	1
D3	539 (1)	6	10	7	6	11	6	1
D4	479 (1)	10	19	11	10	19	10	1
D5	575 (1)	2	3	3	3	5	3	1
D6	143 (1)	2	3	3	2	3	2	1
D7	359 (1)	4	9	5	4	7	4	2
All	2357 (7)	29	52	36	30	53	30	9

Table 9. Experimental results-Example insertions (SC2, SC3, SC4, SC5 scenarios)

Dataset	**Initial Examples**	**Initial Neurules**	**Training Subsets**	**Simple (SI) Method**	**Total Retraining (TR) Method**		**Splitting Tree (ST) Method**	
				Neurules	*Neurules*	*Trained Subsets*	*Neurules*	*Trained Subsets*
D1	142 (2)	3	5	5	3	10 / 5	3	3 / 2
D2	117 (3)	2	3	5	2	9 / 3	2	4 / 2
D3	537 (3)	5	9	8	6	31 / 11	6	5 / 4
D4	475 (5)	7	13	12	10	85 / 19	10	11 / 5
D5	573 (3)	2	3	5	3	11 / 5	3	5 / 3
D6	142 (2)	1	1	3	2	6 / 3	2	3 / 3
D7	357 (3)	3	5	6	4	21 / 7	4	6 / 4
All	2343 (21)	23	39	44	30	173 / 53	30	37 / 23

In Tables 8 and 9, column “Initial Examples” contains the number of the examples of the initial training sets and (in parentheses) the number of the examples that were inserted. For instance, entry “479(1)” at this column (in Table 8) means that the initial training set contains 479 examples and one success example was inserted over those. Column “Initial Neurules” contains the number of neurules produced from the initial

J. Prentzas, I. Hatzilygeroudis, “Incrementally updating a hybrid rule base based on empirical data”, Expert Systems, 24(4), pp. 212-231, 2007.

training sets. Column “Training Subsets” contains the number of training subsets produced from the initial training sets (including the initial training sets themselves). “Neurules” represents the number of final neurules, produced via the three insertion methods. “Trained Subsets” represents the number of training subsets that had to be trained due to the insertion of the success examples.

Table 8 concerns SC1 scenario; each row, except the last one, corresponds to a SC1 case. The collective view of all rows, represented by the last row, corresponds to a SC3 scenario.

From the results in Table 8, it is obvious that SI method increases the number of neurules in NRB compared to the other two methods. The increase may not be considered as significant, if we view each insertion by itself (scenario SC1), but totally, considering all example insertions into the source knowledge (last row, scenario SC3), SI method produces 20% more neurules than ST method. This is graphically illustrated in Fig. 9. This increase in the number of neurules increases the inferences runtime by an average of 4%.

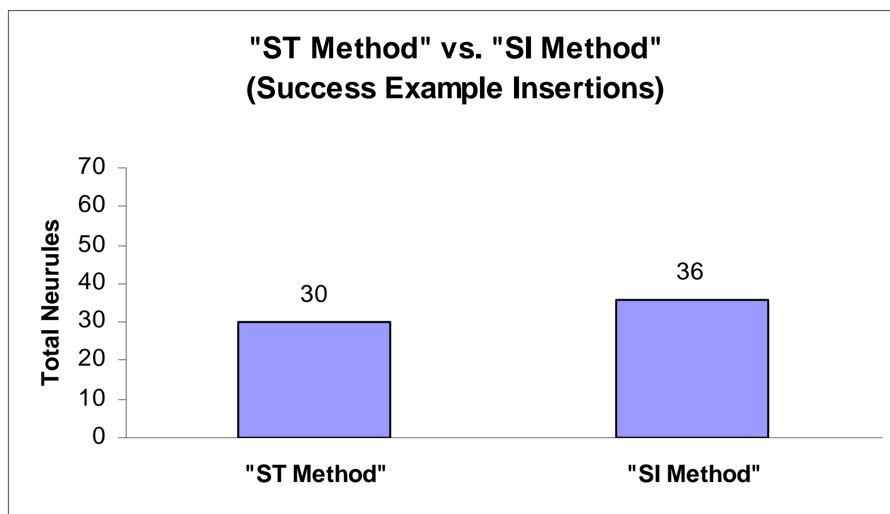

Figure 9. Number of neurules for the cases of Table 8.

Table 10. Runtime results for the cases of Table 8

Dataset	ST method (sec)	TR method (sec)
D1	0.032	0.067
D2	0.001	0.0304
D3	0.004	0.825
D4	0.004	1.44
D5	0.001	0.34
D6	0.001	0.034
D7	0.1	0.45
All	0.143	3.1864

On the other hand, TR method requires more training (and splitting) effort than ST method, especially when the initial training set is split in several subsets. For instance, for datasets D3 and D4, TR method requires training of 11 and 19 training subsets respectively, whereas ST method requires training of only 1 subset in both cases. In the initial training sets that are split in a few training subsets, the difference between

the two methods, in terms of the number of subsets that had to be processed, does not seem to be much enough. However, in those cases, runtime difference between the two methods is significant (see e.g. cases D2, D5 and D6 in Table 10, further on). Considering all example insertions (last row, scenario SC3), TR method requires training of approximately 480% more subsets that ST method (see Fig. 10).

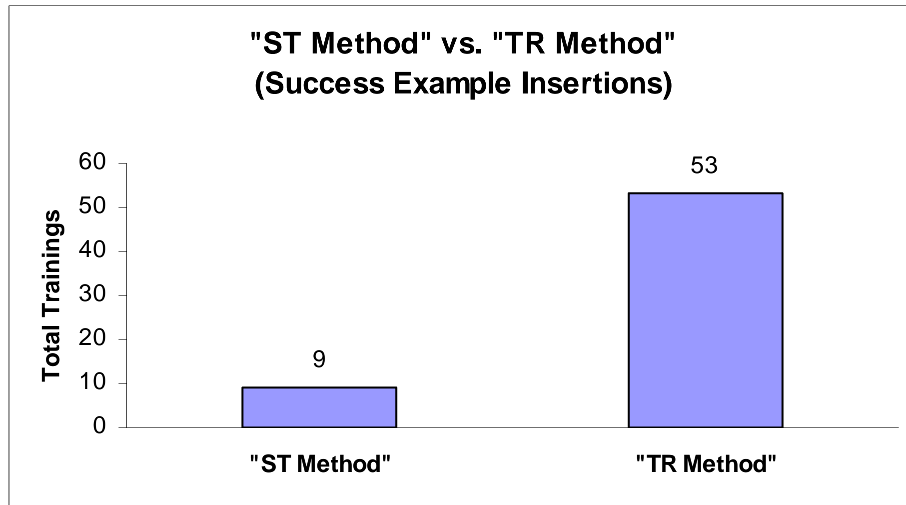

Figure10. Number of trainings for the cases of Table 8.

Table 9 includes SC2 and SC4 cases; each row, except the last one, corresponds to either an SC2 or an SC4 scenario case. The collective view of all rows, represented by the last row, corresponds to either an SC3 or an SC5 scenario. In Table 9, column "Trained Subsets" represents the number of training subsets that had to be trained due to the insertion of the success examples for both scenarios SC2 and SC4. For instance, entry "9/3" at this column (in Table 9) means that the number of training subsets that had to be trained due to the insertion of the success examples were 9 in scenario SC2 and 3 in scenario SC4. The SST method is used in SC4 scenarios.

From the results in Table 9, it is obvious that once again SI method increases the number of neurules in NRB compared to the other two methods. As before, the increase may not be considered as significant, if we view each initial training set separately (scenario SC2). However, considering all example insertions into the SS, SI method produces approximately 46% more neurules than ST method, as illustrated in Fig. 11. Furthermore, this increase in the number of neurules increases the inferences runtime by an average of 9%.

Again, TR method requires more training (and splitting) effort than ST method, especially when the initial training set is split into several subsets. For instance, for datasets D3 and D4, TR method requires training of 28 and 85 subsets, respectively, whereas ST method requires training of only 5 and 11 subsets, respectively. As before, in the initial training sets that are split in a few training subsets, the difference between the two methods does not seem to be significant, but the runtime difference between the two methods is significant (see e.g. cases D5 and D6 in Table 11, later on). Finally, considering all example insertions into the source knowledge (scenarios SC3 and SC5), TR method requires training of approximately 360% and 130% more subsets than ST and SST method respectively (see Fig. 12). Also ST method (scenario SC3) requires training of approximately 60% more subsets than SST (scenario SC5).

Table 10 presents runtime results for ST and TR methods for the example insertions of Table 8 (SC1 cases). It is very clear, from Table 10, that in all cases the runtime required for ST method is less (in the majority of them much less) than the runtime required for TR method. This is due, first, to the fact that ST method requires training of less subsets. Furthermore, ST method avoids processing of training subsets that are higher in the splitting tree, for which training and splitting effort is more expensive than that for subsets lower in the splitting tree (as explained in the introduction of section 5). As expected, runtime differences between the two methods are not proportional to their differences in the number of processed subsets.

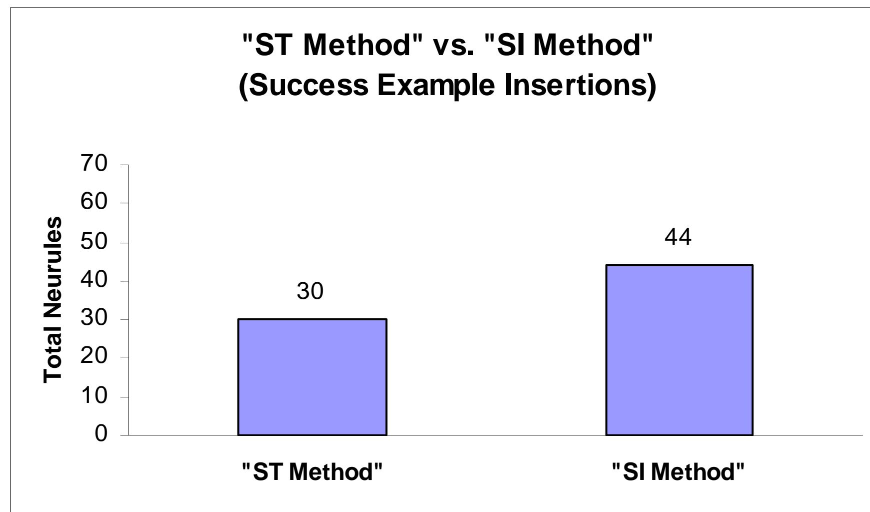

Figure 11. Number of neurules for the cases of Table 9.

Table 11. Runtime results for the cases of Table 9

Dataset	ST method (SC1, SC2, SC3) (sec)	SST method (SC4, SC5) (sec)	TR method (SC1, SC2, SC3) (sec)	TR method (SC4, SC5) (sec)
D1	0.0335	0.0335	0.13	0.067
D2	0.032	0.002	0.09	0.0304
D3	0.305	0.157	2.11	0.825
D4	0.434	0.289	6.10	1.44
D5	0.31	0.155	0.67	0.34
D6	0.034	0.034	0.064	0.034
D7	0.2475	0.147	1.215	0.45
All	1.396	0.8175	10.379	3.1864

Table 11 presents runtime results for ST, SST and TR methods for the cases of Table 9. Table 11 includes runtime results for all scenarios. Each value of the second and fourth column corresponds to an SC1 or SC2 scenario case, except the last one that corresponds to an SC3 scenario case. Each value of the third and fifth column corresponds to an SC4 scenario, except the last one that corresponds to an SC5 scenario case. ST method performs much better than TR method in all cases in scenarios SC1, SC2, SC3. ST method (SC2) performs better than TR method (SC4) in all cases but dataset D2. Furthermore the difference between ST method (SC2) and TR method (SC4) in cases of datasets D5 and D6 is very small. An explanation that can be given for the results regarding datasets D2, D5 and D6 is that the number of training subsets produced from the initial training sets is small and the successive

insertion of each success example affects a large portion of the splitting tree. TR method (SC4) seems to be an alternative to ST method (SC2) in such cases. This is confirmed by the runtime results for datasets D3 and D4. So, ST method (SC2, SC3) performed quite well compared to TR method (SC4, SC5) even though the odds were against it (i.e., consecutive vs. simultaneous insertions).

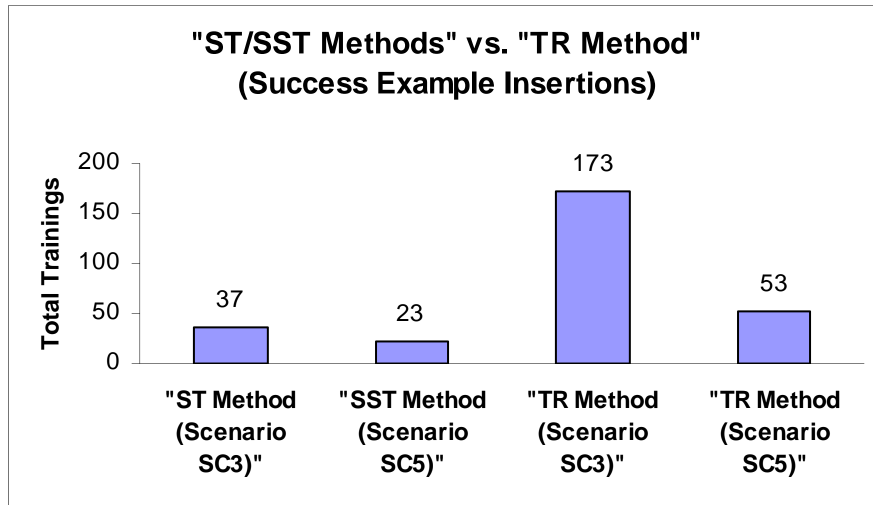

Figure 12. Number of trainings for the insertions shown in Table 9.

Furthermore, as expected, SST method (SC4) performs much better than ST method (SC2, SC3) and obviously TR method (SC2, SC3). In addition, SST method (SC4, SC5) performs much better than TR method (SC4, SC5). Therefore, taking into account the equivalent scenarios, that is ST method (SC2, SC3) vs. TR method (SC1, SC2, SC3) and SST method (SC4, SC5) vs. TR method (SC4, SC5), ST/SST methods perform much better than TR method.

5.3 Results for Failure Example Insertion

Table 12 presents experimental results related to update methods dealing with insertions of failure examples. To produce the results, some failure examples were removed one by one from the initial training sets.

Column “Initial Examples” contains the number of examples of the initial training sets and (in parentheses) the number of examples that were inserted. For instance, entry “142(2)” of this column means that the initial training set contains 142 examples and two failure examples were inserted over those. Column “Initial Neurules” contains the number of neurules produced from the initial training sets. Column “Training Subsets” contains the number of training subsets produced from the initial training sets (including the initial training sets themselves). “Neurules” represents the number of final neurules produced via the two update methods. “Trained Subsets” represents the number of subsets that had to be (re)trained due to the insertion of the failure examples. Table 12 includes SC1, SC2 and SC4 scenario cases; each row, except the last one, corresponds to an SC1, SC2 or an SC4 scenario case. The collective view of all rows, represented by the last row, corresponds to either an SC3 or an SC5 scenario. For SC4 scenarios, the SST method is used.

Table 12. Experimental results-Failure example insertions (SC1, SC2, SC3, SC4, SC5 scenarios)

Dataset	Initial Examples	Initial Neurules	Training Subsets	Total Retraining (TR) Method		Splitting Tree (ST/SST) Method	
				Neurules	*Trained Subsets*	*Neurules*	*Trained Subsets*
D1	142 (2)	3	5	3	10 / 5	3	6 / 3
D2	117 (2)	2	3	2	6 / 3	2	4 / 2
D3	538 (2)	6	11	6	22 / 11	6	12 / 6
D4	475 (5)	10	19	10	95 / 19	10	50 / 10
D5	574 (2)	3	5	3	10 / 5	3	6 / 3
D6	143 (1)	2	3	2	3 / 3	2	2 / 2
D7	357 (3)	4	7	4	21 / 7	4	12 / 4
All	2346 (17)	30	53	30	167 / 53	30	92 / 30

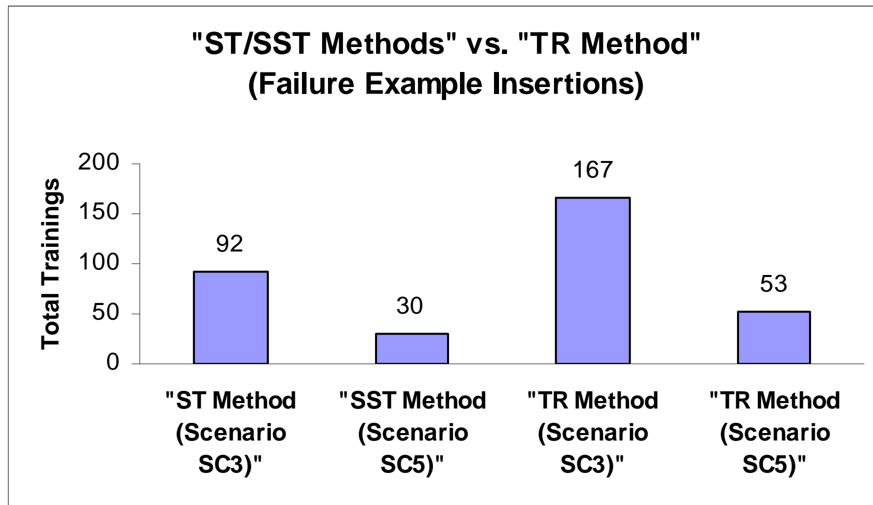

Figure 13. Number of trainings for the cases of Table 12.

Table 13. Runtime results for the cases of Table 12

Dataset	ST method (SC1, SC2, SC3) (sec)	SST method (SC4, SC5) (sec)	TR method (SC1, SC2, SC3) (sec)	TR method (SC4, SC5) (sec)
D1	0.006	0.003	0.128	0.067
D2	0.004	0.002	0.06	0.0304
D3	0.032	0.010	1.65	0.825
D4	0.1	0.032	7.3	1.44
D5	0.013	0.004	0.71	0.34
D6	0.002	0.0006	0.034	0.034
D7	0.012	0.004	1.25	0.45
All	0.169	0.0556	11.132	3.1864

According to the results in Table 12, TR method, as in success example insertions, requires more training (and splitting) effort than ST method (see Fig. 13), especially when the initial training set is split in several subsets. Considering all example insertions into the source knowledge (scenarios SC3 and SC5), TR method requires

training of approximately 81% and 76% more subsets than ST and SST method respectively (see Fig. 13). Also ST method (scenario SC3) requires training of approximately 206% more subsets than SST (scenario SC5).

Table 13 presents runtime results for ST, SST and TR methods for the cases of Table 12. Table 13 includes runtime results for all scenarios. Each value of the second and third column corresponds to an SC1 or SC2 scenario case, except the last one that corresponds to an SC3 scenario case. Each value of the third and fifth column corresponds to an SC4 scenario, except the last one that corresponds to an SC5 scenario case. ST and SST methods perform better than TR method in all cases.

6. Related Work

Our work is related to incremental update of a knowledge-base. Because our knowledge bases consist of hybrid rules, this makes it a unique issue. We are not aware of other similar attempts, i.e. attempts in incrementally updating a hybrid rule base, based on empirical patterns. However, given that neurules can be actually considered as adaline units, which are constructed form empirical example cases, our methods can be considered as incremental learning methods. So, in this section we present works related to incremental learning and place our methods in that context.

The usual assumption that is made with regards to the construction of knowledge bases for classifiers obtaining their knowledge from empirical data is that all training examples are available a priori. However, this is not always the case in all application fields. In certain application fields such as user modeling, robotics and intelligent agents not all training examples are available a priori since a number of them become available over time (Giraud-Carrier 2000). This happens either because the environment changes with time or because the rate at which examples become available may be too slow. To deal with this problem, corresponding update methods should be developed for the various types of classifiers. More specifically, classifiers should be able to learn over time without forgetting what has already been learnt. Broadly used algorithms such as ID3 and back propagation that respectively induce decision trees and train neural networks cannot handle efficiently this situation. In fact, they perform re-learning from scratch for the training set consisting of the new example(s) and old examples. Such an approach is temporally inefficient since previous learning efforts are wasted.

In the literature, the term ‘on-line learning’ is used to describe situations in which learning must take place over time (Maloof and Michalski 2004). Such learning systems may be classified into categories based on two criteria: (a) the portion of the induced hypothesis stored and (b) the percentage of past training examples stored. As far as the induced hypothesis is concerned, the learner may retain all, a part or even none of it. The corresponding categories are referred to as ‘full concept memory’, ‘partial concept memory’ and ‘no concept memory’. Similarly, the learner may retain all, some or none of the past training examples and the corresponding categories are ‘full instance memory’, ‘partial instance memory’ and ‘no instance memory’. Learning systems retaining the entire induced hypothesis are distinguished to ‘temporal batch learners’ and ‘incremental learners’. Temporal batch learners replace the induced hypothesis with a new one resulting from the new example(s) and the past training examples stored (if any). Temporal batch learners thus perform a re-learning task from scratch discarding the previously induced hypothesis. Incremental learning systems on the other hand, use the new example(s) along with possibly retained past

training examples to modify or adjust the induced hypothesis. Incremental learners are therefore more efficient in dealing with new examples as far as the required processing time is concerned.

It should be mentioned that there have been different views about what can be considered as incremental learning. In (Maloof and Michalski 2004) three types of incremental learners are distinguished, according to whether all, some or none of the past training examples are retained. However, a number of researchers consider that incremental learning should not process, retain, or have access to past examples but should take as inputs only the induced hypothesis and the new example(s) (Giraud-Carrier 2000; Fu 1996). More specifically, in such cases, learning is usually carried out on an example-by-example basis. Based on this point of view, only one of the three subcategories of incremental learners specified in (Maloof and Michalski 2004) can be regarded as incremental. Such a view for incremental learning results in both spatial and temporal efficiency but is not always feasible for all types of classifiers. Such a view, in the context of the neuro-symbolic domain, is presented in (Fu 1996). That approach does not require storage of or access to past training examples, but updates a *knowledge based connectionist neural network* (KBCNN) based only on the new example. A representative approach in the context of decision trees is ID4 (Schlimmer and Fisher 1986), an incremental version of ID3. In (Polikar et al. 2001) even more strict criteria for an incremental algorithm are defined, but the presented incremental approach differs from other approaches in that it retains an ensemble of classifiers instead of a single one.

On the other hand, despite the space overhead, retaining past examples may yield advantages compared to the situation in which no past examples are retained. For instance, such cases have been demonstrated in the context of decision trees. ID5R (Utgoff 1989) and ITI (Utgoff, Berkman and Clouse 1997) are incremental versions of ID3 retaining all the training examples in the decision tree and this results in advantages compared to ID4. Retaining past examples does not mean that all of them will have to be reprocessed during updates, to accommodate introduction of a new example. Also, in (Lange and Zeugmann 1996), it is theoretically shown that retaining some of the past examples results in improved learning power compared to the situation in which no past examples are retained.

The update methods developed for neurules retain the entire induced hypothesis and all past training examples. In this way, the update methods for neurules differ from the aforementioned approaches such as (Giraud-Carrier 2000; Fu 1996; Schlimmer and Fisher 1986; Polikar et al. 2001) in that they take as inputs only the induced hypothesis and the new example. According to this point of view the update methods developed for neurules cannot be regarded as incremental learning.

Following the categorization in (Maloof and Michalski 2004), the neurule-based update methods (presented in Section 3) can be classified into different subcategories of the category of learners retaining the entire induced hypothesis (i.e., ‘full concept memory’). More specifically, the SI and ST/SST methods can be classified into the subcategory of incremental learners that retain all past examples (i.e., ‘full instance memory’). Those methods use for training a portion of the past examples along with the new example(s), affecting the corresponding portion of the neurule base and leaving the rest intact. On the other hand, TR method performs temporal batch learning with ‘full instance memory’.

7. Conclusions

J. Prentzas, I. Hatzilygeroudis, “Incrementally updating a hybrid rule base based on empirical data”, Expert Systems, 24(4), pp. 212-231, 2007.

In this paper, we present methods for efficiently updating a hybrid rule base (target knowledge) due to changes to its empirical source knowledge. The hybrid rule base consists of neurules, a type of hybrid rules integrating symbolic rules with neurocomputing. Its empirical source knowledge mainly consists of training examples/patterns. Updates refer to insertion of new training examples and could be considered as a type of incremental learning.

The introduced methods are quite efficient. That is, only the necessary part of the target knowledge has to be retrained and the number of the neurules remains as small as possible. This is achieved by exploiting the notion of closeness, used to handle inseparability in the construction of the target knowledge, and a structure called the splitting tree, which stores information regarding the construction process of the target knowledge.

In this paper, we assume that the existing (past) examples are retained and also a portion of them needs to be reprocessed. An interesting research direction would be to investigate possible development of update methods for neurule bases that do not require (re)processing of existing examples, but perform updates based only on the newly available examples and the existing neurules.

References

R. Andrews, J. Diederich and A. Tickle, "A survey and critique for extracting rules from trained ANN", *Knowledge-Based Systems*, 8, 1995, 373-389.

A.S. d'Avila Garcez, K.B. Broda and D.M. Gabbay, "Symbolic knowledge extraction from trained neural networks: a sound approach", *Artificial Intelligence* 125 (2001) 155-207.

A.S. d'Avila Garcez, K.B. Broda and D.M. Gabbay, *Neural-Symbolic Learning Systems*, Springer-Verlag, Heidelberg, 2002.

L-M Fu and L-C Fu, "Mapping rule-based systems into neural architecture", *Knowledge-Based Systems*, 3, 1990, 48-56.

L-M Fu, "Incremental Knowledge Acquisition in Supervised Learning Networks", *IEEE Transactions on Systems, Man and Cybernetics-Part A: Systems and Humans*, 26, 1996, 801-809.

S. I. Gallant, *Neural Network Learning and Expert Systems*, MIT Press, 1993.

A.Z. Ghalwash, "A Recency Inference Engine for Connectionist Knowledge Bases", *Applied Intelligence*, 9, 1998, 201-215.

C. Giraud-Carrier, "A Note on the Utility of Incremental Learning", *AI Communications*, 13, 2000, 215-224.

L.O. Hall, K. Sanou and S. Romaniuk, "An Encoding of Production Rules in a Connectionist Network", *Journal of Intelligent and Fuzzy Systems*, 4 (1), 1996, 1-18.

I. Hatzilygeroudis and J. Prentzas, "Neurules: Improving the Performance of Symbolic Rules", *International Journal on AI Tools*, 9, 2000, 113-130.

I. Hatzilygeroudis and J. Prentzas, "Constructing Modular Hybrid Rule Bases For Expert Systems*", International Journal on AI Tools*, 10, 2001, 87-105.

I. Hatzilygeroudis and J. Prentzas, “An Efficient Hybrid Rule-Based Inference Engine with Explanation Capability”, *Proceedings of the 14th International FLAIRS Conference*, AAAI Press, 2001, 227-231.

I. Hatzilygeroudis, P.J. Vassilakos and Tsakalidis A., “XBONE: A Hybrid Expert System for Supporting Diagnosis of Bone Diseases”, C. Pappas, N. Maglaveras and J-R Scherrer (Eds), Medical Informatics Europe’97 (MIE’97), IOS Press, 1997, 295-299.

S. Lange, Th. Zeugmann, “Incremental Learning from Positive Data”, *Journal of Computer and Systems Sciences*, 53, 1996, 88-103.

M.A. Maloof, R.S. Michalski, “Incremental learning with partial instance memory”, *Artificial Intelligence,* 154, 2004, 95-126.

K. McGarry, S. Wertmer and J. MacIntyre, “Hybrid neural systems: from simple coupling to fully integrated neural networks”, *Neural Computing Surveys,* 2, 1999, 62-93.

V. Palade, D. Neagu and R. J. Patton (2001). “Interpretation of trained neural networks by rule extraction”, in Bernd Reusch (Ed), “Computional Intelligence: Theory and Applications”, LNCS 2206, 152-161.

R. Polikar, L. Udpa, S.S. Udpa, V. Honavar, “Learn++: An Incremental Learning Algorithm for Supervised Neural Networks”, *IEEE Transactions on Systems, Man and Cybernetics – Part C: Applications and Reviews*, 31(4), 2001, 497-508.

J. Prentzas, I. Hatzilygeroudis and A. Tsakalidis, “Updating a Hybrid Rule Base with New Empirical Source Knowledge”, Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2002), Washington, D.C., USA, November 4-6, 2002, 9-15.

J. Prentzas and I. Hatzilygeroudis, “Rule-based update methods for a hybrid rule base”, *Data & Knowledge Engineering*, 55, 2005, 103-128.

T-S Quah, C-L Tan, R.S. Krishnamurthy and B. Srinivasan, “Towards integrating rule-based expert systems and neural networks”, *Decision Support Systems*, 17 (2), 1996, 99-118.

J.C. Schlimmer, D. Fisher, “A Case Study of Incremental Concept Induction”, Proceedings of the 5th National Conference on Artificial Intelligence, Morgan Kaufmann, 1986, 496-501.

G. Towell and J. Shavlik, “Knowledge-Based Artificial Neural Networks”, *Artificial Intelligence*, 70, 1994, 119-165.

P.E. Utgoff, “Incremental Induction of Decision Trees”, *Machine Learning*, 4, 161-186, 1989.

P.E. Utgoff, N.C. Berkman, J.A. Clouse, “Decision Tree Induction Based on Efficient Restructuring”, *Machine Learning*, 29, 5-44, 1997.

S. Wermter and R. Sun, (eds), *Hybrid Neural Systems*, Springer-Verlag, Heidelberg, 2000.

J. Prentzas, I. Hatzilygeroudis, “Incrementally updating a hybrid rule base based on empirical data”, Expert Systems, 24(4), pp. 212-231, 2007.