Using Precepts to Augment Training Set Learning[1]

Christophe Giraud-Carrier and Tony Martinez

Department of Computer Science, Brigham Young University, Provo, UT 84602

Abstract

The goal of learning systems is to generalize. Generalization is commonly based on the set of critical features the system has available. Training set learners typically extract critical features from a random set of examples. While this approach is attractive, it suffers from the exponential growth of the number of features to be searched. We propose to extend it by endowing the system with some a priori knowledge, in the form of precepts. Advantages of the augmented system are speed-up, improved generalization, and greater parsimony. This paper presents a precept-driven learning algorithm. Its main features include: 1) distributed implementation, 2) bounded learning and execution times, and 3) ability to handle both correct and incorrect precepts. Results of simulations on real-world data demonstrate promise.

1. Introduction

Machine learning seeks to devise mechanisms that machines can use to learn, rather than being explicitly programmed for, real-world applications. If A is an application having inputs $i_1,...,i_n$ and outputs $o_1,...,o_p$, with domains $I_1,...,I_n$, $O_1,...,O_p$, respectively, then $I=I_1\times...\times I_n$ is the input space and $O=O_1\times...\times O_p$ is the output space for A. In this work, A is a function from I to O.

A common approach to machine learning, called *training set learning* (*TSL*) consists of approximating A from a subset of $I\times O$, called the training set. The training set is commonly derived from experimentation. TSL has become popular for both practical and philosophical reasons. It is practical since, for any application A, it is usually possible to come up with a rich training set for A. It is also philosophically attractive because a training set learner extracts the critical features of A from the training set, without human intervention. These critical features are used in turn to approximate A. Empirical studies show that good results can be achieved with TSL [8, 11].

However, TSL has several drawbacks. Training set learners (e.g., backpropagation) are typically slow as they may require many passes over the training set. Also, there is no guarantee that, given an arbitrary training set, the system will find enough good critical features to get a reasonable approximation of A. Moreover, the number of features to be searched is exponential in the number of inputs, and TSL becomes computationally expensive [1]. Finally, the scarcity of interesting positive theoretical results suggests the difficulty of learning without sufficient a priori knowledge.

This paper presents *precept-driven learning* (*PDL*). PDL is intended to overcome some of TSL's weaknesses. In PDL, the training set is augmented by a small set of *precepts*. A pair $p = (i, o)$ in $I\times O$ is called an *example*. A precept is an example in which some of the i-entries (inputs) are set to the special value *don't-care*. An input whose value is not don't-care is said to be *asserted*. If i_j is an input whose value is don't-care in some pair p, then every pair p' obtained from p by asserting i_j to any one of its possible values (in I_j) and leaving the other inputs of p unchanged has the same output as p. Informally, this says that in the context of p, the value of i_j has no effect on the value of the output. The use of the special value don't-care is therefore as a short-hand. A pair containing don't-care inputs *represents* as many examples as the product of the sizes of the input domains of its don't-care inputs.

An example only represents itself (all of its inputs are asserted). Thus, it corresponds to a very specific state of the world. A precept is more general since it represents several examples. A precept reflects some high-level-knowledge about the real-world. It "suggests something advisory and not obligatory communicated typically through teaching" (Webster's Dictionary). Precepts are identical to critical features except that they are given by the teacher and not derived by the algorithm. Also, they

[1] Research funded in part by Novell, Inc. and WordPerfect Corp.

are not necessarily correct, but convey some hint or bias. For instance, when dealing with how weather conditions affect what one wears, a possible precept is: "if it rains, take an umbrella," regardless of the temperature, the day of the week, or any other unnecessary (don't-care) detail.

The majority of learning algorithms do training set learning. Several models exist that allow the presentation of precepts such as ASOCS [4, 5, 6] and ScNets [3]. PDL seeks to overcome ASOCS rigid order-dependency in dealing with inconsistency and to improve generalization.

PDL is psychologically plausible. Seldom do humans learn exclusively from random examples, without ever being exposed to more general (common-sense) rules. For example, regardless of most external circumstances, it does not make sense to put your hand on a red-hot stove. This is a general rule or precept that does not require successive examples of settings involving your hand on the stove, to be acquired. Rather, humans are exposed to precepts taught by others (e.g., "it will burn you"), which, together with a built-in pain mechanism, allow the (almost instantaneous) learning of the more general rule. Foreign-language students first learn English rules (e.g., the plural of a noun is formed by tagging an *s* to the end of the singular form of that noun), and are then exposed to exceptions (e.g., geese instead of gooses). Moreover, PDL speeds up learning by pruning the input space and improves generalization by using precepts.

Section 2 gives the details of a precept-driven learning algorithm (PDLA) capable of handling both correct and incorrect precepts. The approach used follows justice's assertion that individuals are innocent until proven guilty. It consists of always assuming that precepts are correct until enough evidence may be obtained from the training set to decide otherwise. Section 3 describes and analyzes the results of simulations of PDLA against several data sets. Performance is measured: 1) without precepts (pure training set learning), 2) with correct precepts, and 3) with incorrect precepts (contradicting the features of the application). Section 4 concludes the paper.

2. A precept-driven learning algorithm

In this section, we describe a precept-driven learning algorithm (PDLA). Before going into the details of it, we give a few preliminary definitions illustrated by examples. Pairs refer to either examples or precepts.

2.1. Definitions

- Two pairs are *equal* if their input values are equal.
- A pair p is a *superset* of a pair q if p represents all of the examples that q represents, and more. If p is a superset of q, then q is a *subset* of p.
- Two pairs are *discriminated* if they differ in the value of at least one asserted input. If they differ exactly on one asserted input, then there is a *near_match* between them.
- Pairs that are not equal, and do not satisfy the subset, superset or discriminated relations, are said to *overlap*.
- Two pairs are said to be *concordant* if their outputs are equal; otherwise, they are *discordant*.
- A pair p *contradicts* a pair q if p and q are discordant and p is subset or overlap of q.
- *size*: the number of examples the pair represents.
- *num_asserted*: the number of asserted inputs.
- *specificity*: the ratio num_asserted/number of inputs. Examples have specificity=1 (maximally specific), while precepts have specificity<1 (more general).

A pair can be *generalized* in many different ways (see [7] for a list of generalization rules). However, the result is always the same. If a pair p represents a set S of examples, then a pair p' that generalizes p (if such a generalization exists) represents a set S' of examples such that $S \subseteq S'$. Hence, specificity(p')≤specificity(p).

PDLA implements a network of simple nodes. Each node stores a pair p and a *list* of the pairs that contradict p. Nodes have the following attributes:

- *num_matched*: the number of the stored pair's asserted inputs that match the network inputs.
- *activation*: the node's activation defined as *num_matched / num_asserted*.
- *num_exceptions*: the number of examples that contradict the stored pair.

The network architecture is that of a binary tree. The only requirement is for a broadcast and gather facility. Both operations are O(log$n$) where n is the number of nodes in the tree. The tree starts empty and dynamically adapts to new knowledge. During learning, nodes are added and deleted. When a node is added, its position is determined so as to maximize breadth. When a node is deleted, transformations are applied to keep the tree balanced. Hence, the tree is maintained as full as possible. For details on such maintenance mechanisms, see [10].

2.2. Example

Let p1=(<0,0,*,1,*>,<0>), p2=(<0,0,0,1,0>,<1>), p3=(<0,*,*,1,1>,<1>), p4=(<0,0,1,1,1>,<0>). All variables are Boolean. The following illustrates the definitions:

- p1 and p3 are precepts, p2 and p4 are examples.
- p2 and p3 are concordant, p1 and p2 are discordant.
- size(p1) = size(p3) = 4, size(p2) = size(p4) = 1.
- p1 is a superset of p2, p4 is a subset of p3, p1 and p3 overlap, p2 and p4 are discriminated, there is a near_match between p2 and p3.
- p2 contradicts p1, p2 and p4 do not contradict each other since they are discriminated.

- num_asserted(p2) = 5, num_asserted(p3) = 3.
- specificity(p1) = .6, specificity(p4) = 1.

Suppose the current network has three nodes storing p1, p2 and p3, respectively, as shown in Figure 1.

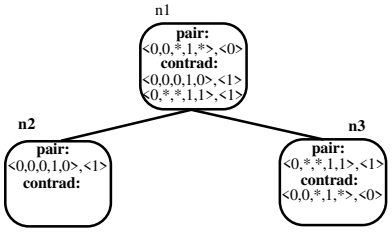

Figure 1 - A Sample Network

Then, for example, num_exceptions(n1)=3, and num_exceptions(n3)=2. When the input is <0,0,1,1,0>, num_matched(n1)=3, activation(n1)=1, and activation(n2)=.8.

2.3. PDLA overview

PDLA can run in two modes: learning and execution. During learning, training pairs are broadcast to all the nodes, and each node reacts to the new pair. It may:

- self-delete,
- update its list of contradictions,
- generalize or cause the new pair to do so,
- attempt to prevent the creation of a new node.

Each node's action is determined by the relation of its stored pair to the training pair. A new node is created if the training pair generalizes or if there does not seem to be a "natural" way for the current network to predict its output. All existing nodes that can generalize do so. If the training pair can generalize and if more than one node causes it to do so, only one generalization is chosen at random, and the node responsible for it self-deletes. Once all training pairs have been presented, exceptions are processed and invalid precepts are deleted.

During execution, the network's inputs are set and each node computes its activation which is a measure of how closely its stored pair matches the network's inputs. The node with highest activation wins and the output part of its stored pair becomes the output of the network. In case of a tie, priority is first given to the most specific pair and then to the one that has the least number of exceptions.

2.4. The learning algorithm

We present PDLA and immediately show an example which explains the details of the algorithm as well as the intuition involved in its design. The network starts empty. Training pairs are presented incrementally and learning proceeds as follows. The *create_flag* keeps track of whether a new node must be created for the training pair and the *modified_flag* keeps track of whether the training pair is candidate for generalization.

Begin PDLA
  For each training pair *q*
    Create node *nn* for *q* with list of contradictions = ∅
    Create_flag = true and modified_flag = false
    For all nodes *n* (storing pair *p*) in the network
      Case superset(*n.p*, *q*)
        If concordant(*n.p*,*q*) Create_flag = false
        Else Update_List_Contradictions(*n*)
      Case equal(*n.p*, *q*)
        If concordant(*n.p*,*q*) Create_flag = false
        Else *n* self-deletes
      Case subset(*n.p*, *q*)
        If concordant(*n.p*,*q*) *n* self-deletes
        Else Update_List_Contradictions(*nn*)
      Case overlap(*n.p*, *q*)
        If discordant(*n.p*,*q*)
          Update_List_Contradictions(*nn*)
          Update_List_Contradictions(*n*)
      Case discriminated(*n.p*, *q*)
        Diff = |num_asserted(*n.p*)-num_asserted(*q*)|
        If concordant(*n.p*,*q*)&near_match(*n.p*,*q*)&diff≤1
          If size(*n.p*)≥size(*q*)&num_asserted(*n.p*)>1
            Drop variable in *n.p*, Create_flag = false
          Else if num_asserted(*q*)>1
            Modified_flag = true
    EndForAll /* all nodes */
    If modified_flag
      Randomly select one of the nodes that set it
      Replace *q* by its corresponding generalization (*q'*)
      Delete the selected node, Retain *nn* (with *q'*)
    Else if create_flag retain *nn*
    Else delete *nn*
  EndForEach /* each training pair */
  Process_Exceptions()
End PDLA

Note that the algorithm automatically handles precepts. All possible relations between all existing nodes and the training pair are checked. Modifications due to dropping variables and updating exceptions are made both to the existing nodes and the new pair. It follows that precepts can be entered in at any time. However, since PDLA is incremental, the order of presentation may affect the outcome. Showing precepts first may speed up learning.

Note also the inherent parallelism of the "For all" loop. All the nodes can execute that loop in parallel.

2.5. Detailed example of PDLA

Consider the following Boolean training set:

p1 = (<0,0,*,1,*>,<0>) p5 = (<1,0,0,1,1>,<0>)
p2 = (<0,0,0,1,0>,<1>) p6 = (<1,1,0,1,1>,<0>)
p3 = (<0,0,1,1,1>,<0>) p7 = (<0,*,*,1,1>,<1>)
p4 = (<1,0,0,1,1>,<1>)

Assume that the pairs are presented to the system in the order p1 through p7. The network starts empty.

Presentation of p1 (<0,0,*,1,*>,<0>). A new node, say n1, is created for p1 and since the network is empty, it is retained. The resulting network is in Figure 2.

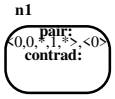

Figure 2 - Network After Presentation of p1

Presentation of p2 (<0,0,0,1,0>,<1>). A new node is created for p2. Node n1 executes the "For all" loop. Its stored pair is superset, discordant. Update_List_Contradictions(n1) executes, causing the addition of p2 to the list of pairs that contradict p1 in n1. The new node, call it n2, is retained. The resulting network is in Figure 3.

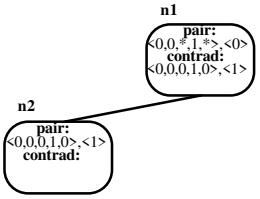

Figure 3 - Network After Presentation of p2

To handle them, PDLA must keep track of exceptions (function Update_List_Contradictions). Each node stores a list of all the pairs that contradict, or are exceptions to, its stored pair. An important issue must be addressed however, namely that of redundancy. If a pair *p* has been contradicted by a pair *q* and is later contradicted by a pair *q′* that is either a subset of, superset of or equal to *q*, then certain precautions must be taken. If *q′* is equal to *q*, discard *q′* (identical exceptions count only once). If *q′* is a subset of *q*, discard *q′* since *q* accounts for the exceptions due to *q′*. If *q′* is a superset of *q*, replace *q* by *q′* (dual of the previous case). Then, num_exceptions is the sum of the amounts of overlap of input space between each pair in the list and the stored pair. The function Process_ Exceptions handles exceptions and is discussed below.

Keeping a list at each node is expensive in terms of storage (O(*n*) where *n* is the size of the training set), and updating the list is computationally expensive (linear in the size of that list). However, unless precepts are really bad (i.e., inconsistent with the training set), the list of contradictions is expected to be small. Moreover, nodes storing such bad precepts are removed from the network.

Presentation of p3 (<0,0,1,1,1>,<0>). A new node is created for p3. Nodes n1 and n2 execute in parallel. The pair in n1 is superset, concordant, so create_flag is set to false. The pair in n2 is discriminated, discordant, so n2 causes no change. The node created for p3 is deleted and the network is unchanged. It is consistent with intuition not to create a new node for a pair that is a subset of and concordant with another pair already in the network.

Presentation of p4 (<1,0,0,1,1>,<1>). A new node, say n3, is created for p4. Nodes n1 and n2 execute in parallel. The pair in n1 is discriminated, discordant, so n1 causes no change. The pair in n2 is discriminated, concordant, but there is no near_match, so n2 causes no change. Node n3 is retained. The resulting network is in Figure 4.

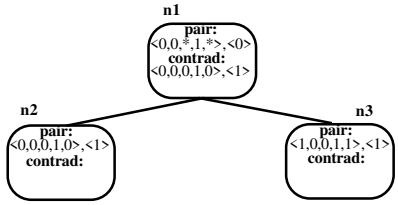

Figure 4 - Network After Presentation of p4

If there does not seem to be a natural way for the current network to predict the output of the training pair, then a new node is created for that pair. In PDLA, natural means that either there exists a node whose pair is concordant, superset, or there is a node causing the new pair to generalize. Neither is true for p4, so n3 is added.

Presentation of p5 (<1,0,0,1,1>,<0>). A new node is created for p5. Nodes n1, n2 and n3 execute in parallel. The pair in n1 is discriminated, concordant, near_match, but Diff=2>1, so n1 causes no change. The pair in n2 is discriminated, discordant, so n2 causes no change. The pair in n3 is equal, discordant so n3 self deletes. The node created for p5 is retained. We call it n3 since the current n3 self deletes. The resulting network is in Figure 5.

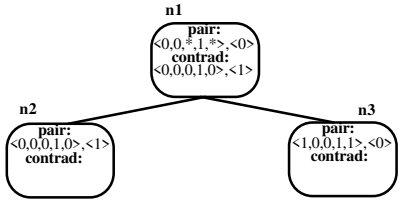

Figure 5 - Network After Presentation of p5

This example shows that PDLA does not handle noise. In this work, *noise* is defined as discordant equal pairs. If two pairs are noisy, only the later one is retained. Another case of noise, which is not addressed here, exists when the application is a distribution rather than a function.

Presentation of p6 (<1,1,0,1,1>,<0>). A new node is created for p6. Nodes n1, n2 and n3 execute in parallel. The pair in n1 is discriminated, concordant, but there is no near_match, so n1 causes no change. The pair in n2 is discriminated, discordant, so n2 causes no change. The pair in n3 is discriminated, concordant, near_match and Diff=0≤1 so generalization takes place. The variable on which the two pairs differ is dropped in the pair stored in

n3 and create_flag is set to false. The node created for p6 is deleted. The resulting network is in Figure 6.

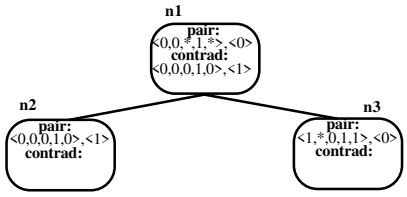

Figure 6 - Network After Presentation of p6

PDLA uses only the dropping condition generalization rule (see [2]). This rule is applied whenever two pairs are concordant, near_match, and one has 0 or 1 more asserted input than the other (variable Diff). The differing input is replaced by the value don't-care in the pair that has the largest size, as long as num_asserted>1. The condition on relative sizes maximizes generalization. The condition on num_asserted ensures that no rule is created whose inputs are all don't-care and that therefore represents all of the input space. Diff measures the "distance" between two pairs and is used to control the level of generalization.

Let p and q be concordant, Boolean, near_match pairs. Diff=0 does not result in generalization, only a smaller network. With Diff=n, generalization results in a pair that is a superset of both p and q as well as of at least 2^n-1 unseen pairs. For n>1, this tends to give very strong generalization (worse with variables other than Boolean). One could argue that incorrect generalization would be dealt with via exception handling. Though this is true, managing the list of exceptions is expensive. The value 1 was empirically found to give the best results.

All existing pairs that can, generalize, thus maximizing overall generalization. If the training pair can generalize, only one variable is dropped, to facilitate implementation.

Presentation of p7 (<0,*,*,1,1>,<1>). A new node, n4, is created for p7. Nodes n1, n2 and n3 execute in parallel. The pair in n1 is overlap, discordant, so Update_List_ Contradictions applies to n1 and n4. The pair in n2 is discriminated, concordant, near_match, but Diff=2>1, so n2 causes no change. The pair in n3 is discriminated, discordant, so n3 causes no change. Node n4 is retained. The resulting network is in Figure 7.

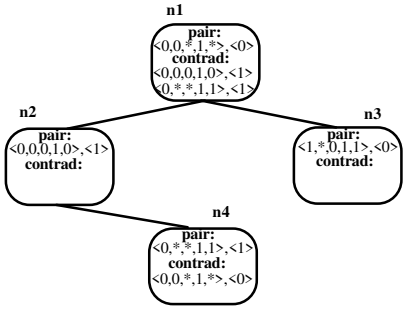

Figure 7 - Network After Presentation of p7

Process_Exceptions(). After all training pairs are seen, exceptions are handled. Precepts are always assumed to be correct until enough evidence is found to refute them. Enough evidence is: num_exceptions(n) > 1/2 size($n.p$). The threshold of half seems reasonable, and is confirmed by empirical results. A smaller threshold may cause the deletion of general rules in favor of their exceptions, and a larger threshold may not delete enough incorrect rules.

Here, n1 is deleted since num_exceptions(n1)=3>2= size(n1.p)/2; n4 is not deleted since num_exceptions(n4) =2≤size(n4.p)/2. The final network is in Figure 8.

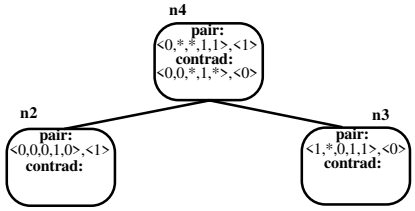

Figure 8 - Final Network

2.6. The execution mode

During execution, the network is presented unknown inputs and must guess the correct output. Each node computes its activation, and a winner is found as follows:

Begin Exec
  If there is a single node with highest activation, it wins
  Else if two nodes n and m are tied
    If specificity($n.p$)>specificity($m.p$) n wins
    Else if specificity($n.p$)=specificity($m.p$)
      If num_exceptions(n)<num_exceptions(m) n wins
      Else m wins
    Else m wins
  Network output = Output of pair in winning node
End Exec

Recall that activation measures closeness of match. When two nodes are tied for highest activation, priority is given to the one whose stored pair is more specific. In particular, if a new pair is a subset of a general rule, but yet is equal to a more specific pair, then it is one of the rule's exceptions and should be recognized as such. If the new pair is not equal to one of the seen exceptions, then its output will be determined by the general rule. This priority scheme makes it possible to deal with exceptions, but also has the tendency to learn noise.

When two nodes are tied and have the same specificity, priority is given to the one with fewer exceptions, thus the output is set to the value that is most likely to be correct.

3. Simulation results

PDLA was tested on several applications, drawn from the Irvine machine learning database [9], with two modifications. All real-valued attributes are made discrete

(using a K-means algorithm on each input), and all don't-know values are treated as don't-care values.

For each data set, results are reported with and without precepts. The number and generality of the precepts vary. Precepts were obtained from the author's common sense knowledge about the application's domain (Lenses, Zoo, Hepatitis) or from a priori knowledge about the problems (Iris, Voting, Credit-Screening, Chess). Tests were run with both correct and incorrect precepts. The incorrect precepts were obtained from the correct ones by changing the value of their outputs.

Each of the reported results is an average over 10 runs of PDLA. In each run, the training set and test set are randomly regenerated. Precepts remain unchanged and are presented first. Both training set and test set contain one half of the examples in the complete data set. Results are summarized in Table 1. The numbers following the percentage of correct guesses, or predictive accuracy (PA) correspond respectively to training set size (TS), including precepts where applicable, and final network size (NS) after the learning phase (in number of nodes).

	No precepts			**Good precepts**			**Bad precepts**		
Application	PA	TS	NS	PA	TS	NS	PA	TS	NS
lenses	67.5	12	6	78.3	13	4	50.8	13	6
zoo	95.8	45	22	95.8	46	19	85.1	46	23
voting	88.5	217	73	91.0	218	31	82.2	218	72
iris	92	75	33	92.7	81	22	68	81	38
hepatitis	77.1	77	71	79.5	79	61	61.4	79	72
credit-scr.	79.4	345	343	83.3	346	203	50.5	346	304
chess-ends	90.5	1598	937	92.5	1599	770	90.5	1599	937
mushroom	100	4062	1499	(unnecessary)			(unnecessary)		

Table 1 - PDLA-Simulation Results

Table 1 shows that the addition of simple precepts improves generalization performance and significantly reduces the size of the network (greater parsimony). On average, PA is increased by 4.3%, while NS is reduced by 30%. Table 1 also shows that PDLA has some ability to deal with bad precepts. In most cases, the "recovery" is good, but is obtained at the expense of a larger network.

4. Conclusion

This paper introduces precept-driven learning and describes a precept-driven learning algorithm (PDLA). Results of simulations on several data sets are reported. PDLA's main features include:

- Distributed implementation facilitating the use of massive parallelism and self adaptation,
- Bounded learning and execution times complexity,
- Use of precepts resulting in faster learning, increased parsimony, and improved generalization,
- Ability to deal with incorrect precepts.

Recent efforts at extending PDLA include rebroadcasting modified pairs, making the system more inductive by only processing pairs that are not correctly classified, and replacing bad precepts by new ones (e.g., toggle output). Future research should focus on noise handling, improved generalization schemes, dealing with missing inputs, and alternatives in handling exceptions.

References

[1] Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M.K. Occam's Razor. *Information Processing Letters*, 24, 1987, 377-380.

[2] Dietterich, T.G., and Michalski, R.S. A Comparative Review of Selected Methods for Learning from Examples. In Michalski, R.S., Carbonell, J.G., and Mitchell, T.M., (Eds.), *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann Publishers, Inc., Vol. I, 1983, Chapter 3.

[3] Hall, L.O., and Romaniuk, S.G. A Hybrid Connectionist, Symbolic Learning System. In *Proceedings of the 8th National Conference on Artificial Intelligence*, 1990, 783-788.

[4] Martinez, T.R., and Vidal, J.J. Adaptive Parallel Logic Networks. *Journal of Parallel and Distributed Computing*, Vol. 5, No. 1, 1988, 26-58.

[5] Martinez, T.R., and Campbell, D.M. A Self-Adjusting Dynamic Logic Module. *Journal of Parallel and Distributed Computing*, Vol. 11, No. 4, 1991, 303-313.

[6] Martinez, T.R., and Campbell, D.M. A Self-Organizing Binary Decision Tree for Incrementally Defined Rule Based Systems. In *IEEE Transactions on System, Man, and Cybernetics*, Vol. 21, No. 5, 1991, 1231-1238.

[7] Michalski, R.S. A Theory and Methodology of Inductive Learning. *Artificial Intelligence*, *20*, 1983, 111-161.

[8] Mooney, R., Shavlik, J., Towell, G., and Gove, A. An Experimental Comparison of Symbolic and Connectionist Learning Algorithms. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, 1989, 775-780.

[9] Murphy, P.M., and Aha, D.W. *UCI Repository of machine learning databases*. Irvine, CA: University of California, Department of Information and Computer Science, 1992.

[10] Rudolph, G., and Martinez T.R. An Efficient Static Topology for Modeling ASOCS. In Kohonen, et al., (Eds.), *Artificial Neural Networks*. Elsevier Science Publishing, 1991, 729-734.

[11] Sejnowski, T.J., and Rosenberg, C.R. Parallel Networks that Learn to Pronounce English Text. *Complex Systems*, 1, 1987, 145-168.