Learning Non-Linearly Separable Boolean Functions With Linear Threshold Unit Trees and Madaline-Style Networks

Mehran Sahami

Department of Computer Science
Stanford University
Stanford, CA 94305
sahami@cs.Stanford.EDU

Abstract

This paper investigates an algorithm for the construction of decisions trees comprised of linear threshold units and also presents a novel algorithm for the learning of non-linearly separable boolean functions using Madaline-style networks which are isomorphic to decision trees. The construction of such networks is discussed, and their performance in learning is compared with standard Back-Propagation on a sample problem in which many irrelevant attributes are introduced. Littlestone's Winnow algorithm is also explored within this architecture as a means of learning in the presence of many irrelevant attributes. The learning ability of this Madaline-style architecture on non-optimal (larger than necessary) networks is also explored.

Introduction

We initially examine a non-incremental algorithm that learns binary classification tasks by producing decision trees of linear threshold units (LTU trees). This decision tree bears some similarity to the decision trees produced by ID3 (Quinlan 1983) and Perceptron Trees (Utgoff 1988), yet it seems to promise more generality as each node in our tree implements a separate linear discriminant function while only the leaves of a Perceptron Tree have this generality and the remaining nodes in both the Perceptron Tree and the trees produced by ID3 perform a test on only one feature. Recently, Brodley and Utgoff (1992) have also shown that the use of multivariate tests at each node of a decision tree often provides greater generalization when learning concepts in which there are irrelevant attributes.

Furthermore, as presented in (Brent 1990), we show how such an LTU tree can be transformed into a three-layer neural network with two hidden layers and one output layer (the input layer is not counted) and can often be trained much more quickly than the standard Back-Propagation algorithm applied to an entire network (Rumelhart, Hinton, & Williams 1986). After examining this transformation, a new incremental learning algorithm, based on a Madaline-style architecture (Ridgway 1962, Widrow & Winter 1988), is presented in which learning is performed using such three-layer networks. The effectiveness of this algorithm is assessed on a sample non-linearly separable boolean function in order to perform comparisons with the LTU tree algorithm and a similar network trained using standard Back-Propagation.

Being primarily interested in functions in which many irrelevant attributes exist, we also explore the performance of the Winnow algorithm (Littlestone 1988, 1991) (which has proven effective in learning linearly separable functions in the presence of many irrelevant attributes) within the Madaline-style learning architecture. We contrast how it performs in learning our sample non-linearly separable function with the classical fixed increment (Perceptron) updating method (Duda & Hart 1973). We also examine the effectiveness of such learning procedures in "non-optimal" Madaline-style networks, and comment on possible future extensions of this learning architecture.

The LTU Tree Algorithm

The tree building algorithm is non-incremental requiring that the set of all training instances, S, be available from the outset.[1] We begin with the root node of the tree and produce a hyperplane to separate our training set using any means we wish (in our trials, Back-Propagation was applied to one node to produce a single separating hyperplane) into the sets S_0 and S_1, where S_i (i = 0, 1) indicates the set of instances classified as i by the separating hyperplane. If there are instances in S_0 which should be classified as 1 (called "incorrect 0's") we then create a left child node and recursively apply the algorithm on the left child using S_0 as the training set. Similarly, if any instances in S_1 should be classified as 0 ("incorrect 1's") we create a right child node and again recursively apply our algorithm on the right child using S_1 as the training set. Thus the algorithm normally terminates when all of the instances in the original training set, S, are correctly classified by our tree.

The classification procedure using the completed tree requires us to simply begin at the root node and determine whether the given instance is classified as a 0 or 1 by the hyperplane stored there. A classification of 0 means we follow the left branch, otherwise we follow the right, and recursively apply this procedure with the hyperplane stored at the appropriate child node. The classification given at a leaf node in the tree is the final output of the classification procedure. Note that the leaves in this decision tree do not

[1]Notation and naming conventions in the description of the LTU tree algorithm are from Brent (1990).

classify all instances into one labeling, rather the classification for the instance is the result of applying the linear discriminator stored in the leaf node.

For our experiments, certain (reasonable) limiting assumptions were placed on the building of such LTU trees in order to prevent needlessly complex trees, thereby helping to improve generalization and reduce the algorithm's execution time. These included setting a maximum tree depth of 10 layers and tolerating a certain percentage of error in each individual node. This toleration condition was set after some empirical observations which indicated that given some number of similarly classified instances in a node, *n*, a certain percentage of erroneous classifications, *E*, would be acceptable (thus precluding further branching for that particular classification from the node). These values are as follows:

- if $n \leq 25$ then $E = 25\%$
- if $n > 25$ & $n \leq 100$ then $E = 12\%$
- else $E = 6\%$

Initial testing was performed within this LTU tree architecture using a variety of methods for learning the linear discriminant at each node of the tree (Sahami 1993). Wishing to minimize the number of erroneous classifications made at each node in the tree, Back-Propagation appeared to be the most promising of these weight updating procedures. While this heuristic of minimizing errors at each node can occasionally produce larger than optimal trees[2], it generally produces trees of optimal or near-optimal size, and was shown to produce the smallest trees on a number of sample functions when compared with other weight updating procedures. Since we are only allowed to store one hyperplane at each node (and not an entire network, although this might be an interesting angle for further research) we apply the Back-Propagation algorithm to only one unit at a time. To make this unit a linear *threshold* unit, a threshold is set at 0.5 *after* training is completed (this threshold is not used during training). Thus the output of the unit trained with Back-Propagation is given by:

$$O_{LTU_n} = \begin{cases} 1 & O_n \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

$$O_n = \frac{1}{1 + e^{-N_k}}, \text{ where } N_k = \vec{w}_k^t \vec{x}_k + \theta_{k-1}$$

where O_n is the actual real valued output of the *n*th trained unit on any instance and O_{LTUn} is the output of our "linear threshold unit." θ represents the "bias" weight of the unit.

The updating procedure used in training each node is:

[2]An optimal tree would contain the minimum number of linear separators (nodes) necessary to successfully classify all instances in the training set, S.

$$\vec{w}_{k+1} = \vec{w}_k + \Delta\vec{w}_k + (momentum)\Delta\vec{w}_{k-1}$$
$$\Delta\vec{w}_k = (lrate * \vec{x}_k * O_k * (1 - O_k) * (d - O_k))$$
$$\text{where } O_k = \frac{1}{1 + e^{-N_k}}, \ N_k = \vec{w}_k^t \vec{x}_k + \theta_{k-1}$$
$$\text{and } \theta_{k+1} = \theta_k + \Delta\theta_k$$
$$\Delta\theta_k = (lrate * O_k * (1 - O_k) * (d - O_k))$$

Where **w** is the weight vector being updated and **x** is a given instance vector. We set *lrate* = 1.0 and *momentum* = 0.5 in our experiments.

There are many possible extensions to this LTU tree-building algorithm including irrelevant attribute elimination (Brodley & Utgoff 1992), producing several hyperplanes at each node using different weight updating procedures and selecting the hyperplane which causes the fewest number of incorrect classifications, using Bayesian analysis to determine instance separations (Langley 1992), post-processing of the tree to reduce its size, etc. These modifications are beyond the scope of this paper however, and generally are only fine tunings to the underlying learning architecture which is not changed by them.

Creating Networks From LTU Trees

The trees which are produced by the LTU tree algorithm can be mechanically transformed into three-layer connectionist networks that implement the same functions. Given an LTU tree, *T*, with *m* nodes, we can construct an isomorphic network containing the *m* nodes of the tree in the first hidden layer (each fully connected to the set of inputs). The second hidden layer consisting of *n* nodes (*AND* gates), where *n* is the number of possible distinct paths between the root of *T* and a leaf node (a node without two children). And the output layer merely being an *OR* gate connected to all *n* nodes in the previous layer. The connections between the first and second hidden layers are constructed by traversing each possible path from the root to a leaf in the tree *T*, and at each node recording which branch was followed to get to it. Thus each node in the second hidden layer represents a single distinct path through *T* by being connected to those nodes in the first layer which correspond to the nodes that were traversed along the given path. Since the nodes in the second hidden layer are merely *AND* gates, the inputs coming from the first hidden layer must first be inverted if a left branch was traversed in *T* at the node corresponding to a given input from the first hidden layer. Two examples are given below.

As pointed out in (Brent 1990), it is more efficient to do classifications using the tree structure than the corresponding network since the only computations which must be performed are those which lie on a single path from the root of the tree to a leaf. Conveniently, when we later examine how to incrementally train a network which corresponds to an LTU tree, we may then transform the trained network into a decision tree to attain this computational benefit during classification.

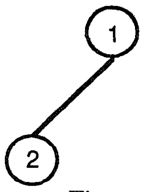


Figure 1

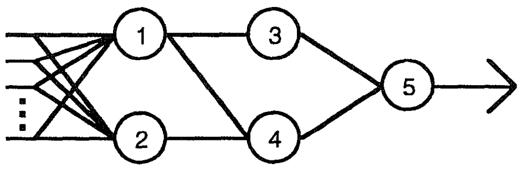


Figure 2

Figure 1 shows a two node tree produced by the LTU tree algorithm, while Figure 2 shows the corresponding network after performing the transformation described above. Nodes 1 and 2 in Figure 1 correspond directly to nodes 1 and 2 in Figure 2. Node 3 simply has the output of node 1 as its input (since there is a path of length 1 in the tree from the root to node 1 which is considered a leaf.) Node 4 is a conjunct of the *inverted* output of node 1 (since we must follow the left branch from node 1 to reach node 2 in the tree) and the output of node 2. Node 5 is simply an *OR* gate.

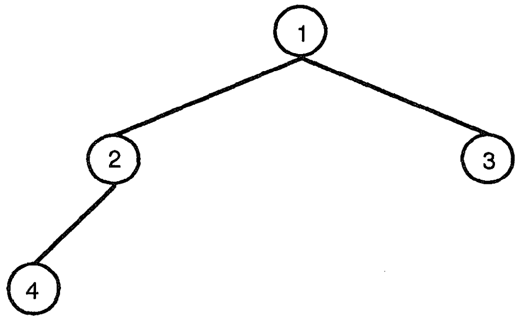


Figure 3

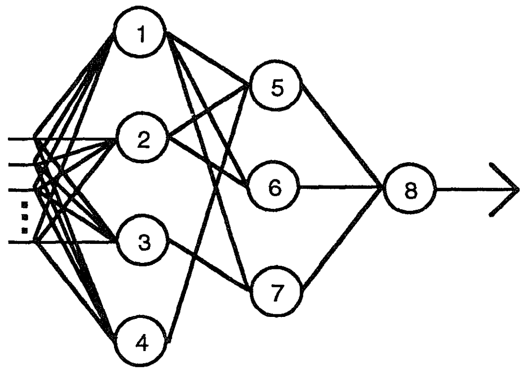


Figure 4

Figure 3 shows a more complex tree produced by the LTU tree algorithm, and Figure 4 represents the corresponding network. Nodes 1, 2, 3, and 4 in Figure 3 correspond directly to the same nodes in Figure 4. In Figure 4, node 5 represents the path 1-2-4 in the tree, with the *inverted* output of node 1, *inverted* output of node 2 and output of node 4 as inputs. Node 6 represents the path 1-2 (as node 2 in the tree is also considered a leaf) with the *inverted* output of node 1 and the output of node 2 as inputs. Node 7 corresponds to the path 1-3 and has the outputs of nodes 1 and 3 as inputs. Again, node 8 is simply a disjunction of the outputs of nodes 5, 6 and 7.

Madaline-Style Learning Algorithm

The updating strategy in this Madaline-style architecture is based upon modifying the weight vectors in the first hidden layer of nodes by appropriately *strengthening* and *weakening* them based on incorrect predictions by the network. We also make use of knowing the structure of the LTU tree, *T*, which corresponds to the network we are training. When an instance is incorrectly classified as a 0, we know that no nodes in the second hidden layer corresponding to a leaf in *T* fired. Thus we look for the node corresponding to a leaf node in *T* which is closest to threshold and strengthen it. We also examine any nodes corresponding to non-leaf nodes in *T* that we would know exists along the path from the root of *T* to the given leaf node closest to threshold. If these nodes were over threshold but the given leaf is down their left child in *T*, then the node in the network corresponding to the particular non-leaf node in *T* is weakened. Similarly if the node corresponding to a non-leaf node in *T* was under threshold, but the leaf node is on a path down its right child in *T*, then the node in the network corresponding to the non-leaf node in *T* is strengthened. When an instance is misclassified as a 1, we simply find the node in the second hidden layer of the network which misfired (there can only be one) and weaken all nodes which are inputs to it and also correspond to leaf nodes in *T*. In the case of the network in Figure 2, this translates in to the following updating procedure:

On a misclassified 0, determine if node 1 or node 2 is closer to threshold:
• If node 1 is closer to threshold, then strengthen node 1, else strengthen node 2.

On a misclassified 1, only node 3 or 4 (but not both) misfired in this case:
• If the output of node 3 is 1,then weaken node 1, else weaken node 2.

How nodes are strengthened and weakened is based upon what learning method was being used on the Madaline-style networks. Both the classical fixed increment (referred to simply as Madaline below) and Littlestone's Winnow algorithm (referred to as Mada-winnow) were employed in our tests as follows:

Algorithm	Updating Method
Fixed Increment	Strengthen:
(Madaline)	$\vec{w}_{k+1} = \vec{w}_k + \vec{x}$
	Weaken:
	$\vec{w}_{k+1} = \vec{w}_k - \vec{x}$
Winnow	Strengthen:
(Mada-winnow)	$\vec{w}^{i}_{k+1} = \alpha^{\vec{x}^{i}}\left(\vec{w}^{i}_{k}\right)$
	Weaken:
	$\vec{w}^{i}_{k+1} = \beta^{\vec{x}^{i}}\left(\vec{w}^{i}_{k}\right)$

Where w is the weight vector (w^i is the *i*th component of w) at the node being modified and x is the instance vector which was misclassified. Note that α=2.0 and β=0.5 (Winnow also uses a fixed threshold which was set to 4.0 in our initial experiments).

Experimental Results

In testing the LTU tree algorithm and the corresponding network for their abilty to learn, a non-linearly separable 5-bit boolean function was used. This function was defined as:

$$\left(\sum_{i=1}^{5}\vec{x}^{\,i} \le 1\right) \vee \left(\sum_{i=1}^{5}\vec{x}^{\,i} \ge 4\right)$$

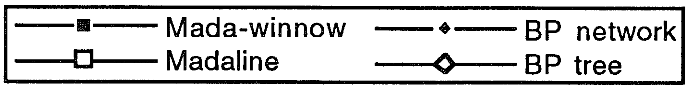
This function, effectively being the disjunction of two *r*-of-*k* threshold functions, is not linearly separable, but can be optimally learned using *two* hyperplanes to separate the instance space. Thus in testing our various learning methods on this function, we compare the LTU tree algorithm against training networks configured similarly to Figure 2 (as this is the optimal size network to learn the given function). In training the networks, we compare standard Back-Propagation applied to the *entire* network (using preset fixed weights in the second hidden and output layers to simulate the appropriate *AND* and *OR* gates) against our novel Madaline-style learning method (discussed above). Note that our learning procedure is effectively only learning the separating hyperplanes in the first hidden layer of the network (corresponding to learning the nodes of an LTU tree).

On a technical note, the instance vectors presented to both the LTU tree and Back-Propagation applied to an *entire* network include the original boolean vector (comprised of 1's and 0's) with the complements of the original vector to create a "double length" instance vector (as preliminary testing showed that the use of complements helped improve learning performance with these algorithms.) In the Madaline-style tests, the instance vectors presented when using fixed increment updating were composed of 1's and -1's without the addition of complements, whereas when using Winnow the instance vectors were similar to those with the LTU tree (complementary attributes were added).

The number of instances presented for training, as well as the number of dimensions in the input vector were varied. Note that only the first 5 bits of the instance vector are relevant to its proper classification and the added bits are simply random, irrelevant attributes. The dimensions given in the graphs below measure the size of the original instance vector (not including complementary attributes). The graphs below represent 5 test runs on each algorithm in each case. Testing is done on an independent, randomly generated set of instances, numbering the same as the training set. The "% error (average)" refers to the percentage of errors made during testing by each algorithm over the 5 test runs. The "% error (best)" refers to the smallest percentage of errors made during testing over the 5 test runs.

We see that, in the average case (Figure 5), when trained using 1000 instances (which are each seen only once), the Madaline network (using fixed-increment updating) outperforms all other algorithms as the number of irrelevant attributes is increased. The LTU tree (called BP tree here) performs without errors up to 15 dimensions (during which time it was consistently producing optimal trees of 2 nodes) and quickly begins to degenerate in performance as the trees it produces get larger due to poor separating hyperplanes being produced at each node. Not surprisingly, it is at this same point when using Back-Propagation over an entire network also begins to degenerate quickly leading us to realize that the network is getting too small to properly deal with irrelevant attributes. Mada-winnow also performs very erratically, due primarily to seeing too few instance vectors to settle into a good "solution state." The best case analysis (Figure 6) indicates a simple linear increase in the number of errors made by Madaline (caused by a linear increase in the sum of weights from irrelevant attributes) as opposed to an erratic increase indicating that the boolean function was not learned. Similarly, Mada-winnow seems to be capable of learning the function up to 35 dimensions and quickly degenerates indicating that learning is not effectively taking place, as opposed to occasional misclassifications caused by added irrelevant attribute weights. We find the BP network still unable to learn beyond 15 dimensions, while the BP tree is still effective up to 30 dimensions.

When we examine the results of using 3000 training instances (each of which is seen once), the effectiveness of the Madaline-style architecture becomes much more clear. In the average case (Figure 7) we still find the standard BP network degenerating after 15 dimensions. However, we see extremely low error rates in Madaline all the way through, indicating that not only has the target function been learned, but the effect of irrelevant attribute weights has also been minimized. Moreover, we find that Mada-winnow is successful in learning the target function with instances up to 35 dimensions in length before its predictive accuracy begins to fall. Similarly, the BP tree is effective for instances up to 40 dimensions before once



Trained using 1000 randomly generated instances

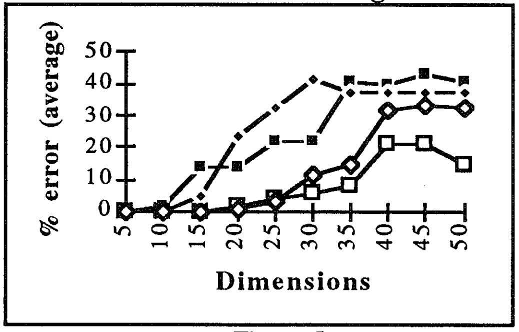


Figure 5

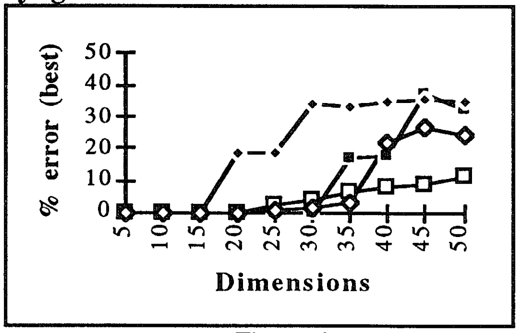


Figure 6

Trained using 3000 randomly generated instances

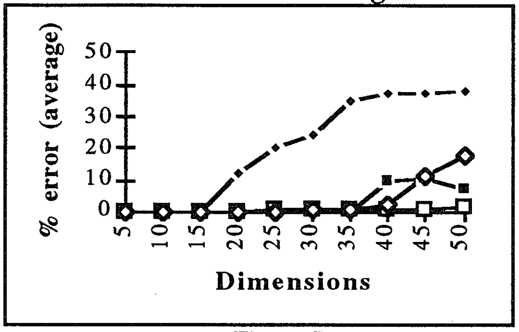


Figure 7

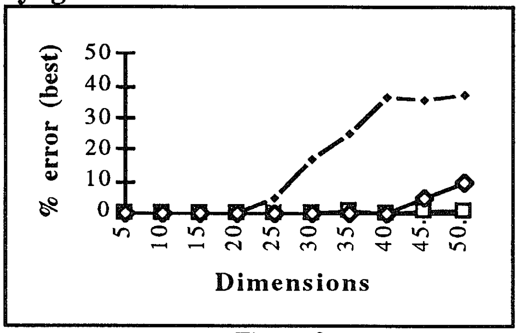


Figure 8

again tree sizes grow too large as the linear separators at each node provide poorer splits. In the best case (Figure 8) we see the most striking results as Madaline still continues a very low error rate, and Mada-winnow has 0% errors over the entire range of dimensions tested! This would indicate that by training a number of such Mada-winnow networks and using cross-validation techniques to determine which has the highest predictive accuracy, we can learn non-linearly separable boolean functions with an extremely high degree of accuracy even in the presence of many irrelevant attributes. This of course does require some knowledge as to what network size would provide the best results, but initially running the LTU tree algorithm on our data set could provide us with good ballpark approximations for this.

Non-Optimal Networks

Having seen the predictive accuracy of the Madaline-style networks in learning when the optimal network size[3] was known, it is important to get an idea for the accuracy of such networks when they are non-optimal. In examining the effects of using a network that is larger than necessary, the network in Figure 4 was used to learn the same 5-bit non-linearly separable problem. The updating procedure for this network is described below:

On a misclassified 0, determine if node 2, 3 or 4 is closest to threshold:

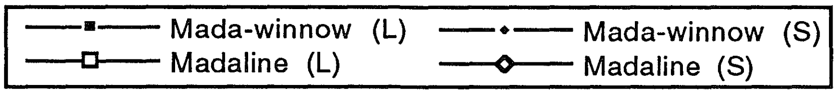
- If node 2 is closest to threshold, then strengthen node 2 and if node 1 is over threshold then weaken node 1.
- If node 3 is closest to threshold, then strengthen node 3 and if node 1 is not over threshold then strengthen node 1.
- If node 4 is closest to threshold, then strengthen node 4 and if node 1 is over threshold then weaken node 1.

On a misclassified 1, determine if node 5, 6 or 7 misfired:

- If the output of node 5 is 1, then weaken node 4.
- If the output of node 6 is 1, then weaken node 2.
- If the output of node 7 is 1, then weaken node 3.

Now we compare the previous results of Madaline and Mada-winnow using the smaller network, denoted (S), with the larger network, denoted (L). Again looking at the average of 5 test runs on 1000 training instances (Figure 9), we see that the performance of both Madaline and Mada-winnow are worse when learning using a larger network (as

[3]The notion of optimal network size stems from the transformation of an optimal LTU tree.



Trained using 1000 randomly generated instances

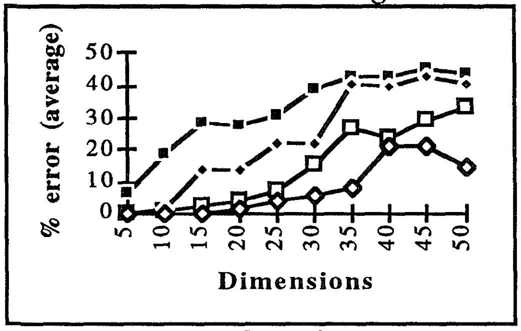


Figure 9

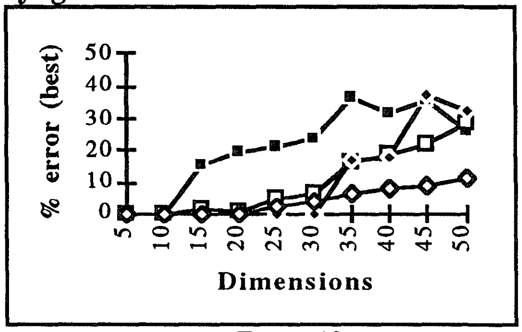


Figure 10

Trained using 3000 randomly generated instances

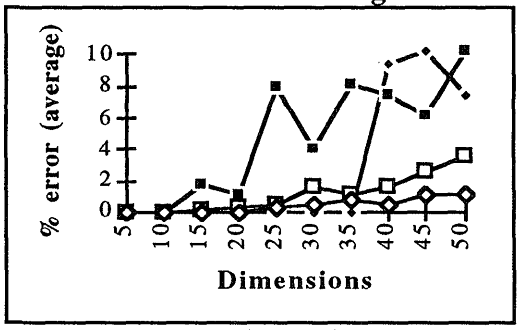


Figure 11

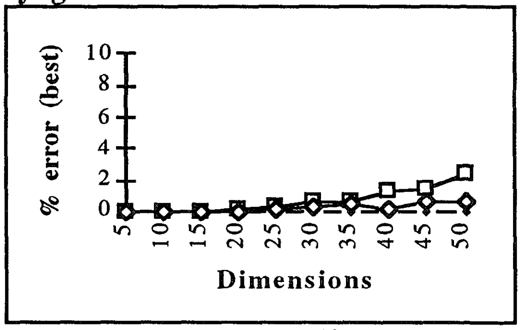


Figure 12

we would expect, since there is greater possibility for confusion among which nodes to update). This is also seen in the best case graph (Figure 10) where we still see the erratic behavior of learning using the Mada-winnow (L) algorithm, which cannot properly learn the target function even with only a few irrelevant dimensions. The Madaline (L) algorithm still holds some promise as it maintains a relatively low error rate until about the 30 dimension mark before it too begins to quickly degenerate in its predictive ability.

Again the most striking differences are seen when examining the graphs of learning runs using 3000 training instances. Noting that the "% error" scale on Figures 11 and 12 is much less than the previous figures (to make the graph more readable), we see that in the average case, while Mada-winnow (L)'s behavior is still erratic (caused by the way the Winnow algorithm greatly modifies weights between each update, leading to instability in the resultant weight vector when training ceases), but the error rate stays below 10%. Moreover, Madaline (L) only shows a small linear decrease in its predictive ability over the entire graph, reflecting again that the target function was effectively learned and misclassifications are arising from the cumulative sum of small irrelevant attribute weights. Finally, Figure 12 shows the most impressive results. First, Madaline(L) has only a slightly higher error rate that Madaline (S). And more impressively, the Mada-winnow (L) algorithm is able to maintain 0% error over the entire range of irrelevant attributes, reflecting that network size is not entirely crucial for effectively learning within this paradigm. An examination of the weights in the larger network indicated that, in fact, two nodes in the first hidden layer contained the appropriate hyperplanes required to learn the target function and the other two nodes had somewhat random but essentially "unused" weights in terms of instance classification.

It is important to note that the fixed threshold used with the Winnow algorithm was dependent on the number of irrelevant attributes in the instance vectors presented. This reflects a problem inherent in the Winnow algorithm (in which threshold choice can have a large impact upon learning) and is not a shortcoming of the Madaline-style architecture.

Future Work

There is still a great deal of work that needs to be done in examining and extending both the LTU tree and the Madaline-style learning algorithms. In terms of the LTU tree, new methods for finding better separating hyperplanes as well as the incorporation of post-learning pruning techniques would be very helpful in determining proper network size both for Madaline-style and standard neural networks. As for the Madaline-style networks, clearly more work needs to be done in examining larger networks and learning more complex functions. Another interesting problem arises in looking at methods to prune the network *during* training to produce better classifications. Also theoretical measures are needed for the number of training instances to present for adequate learning.

Acknowledgments

The author is grateful to Prof. Nils Nilsson, without whose ideas, guidance, help and support, this work would never have been done. Additional thanks go to Prof. Nilsson for reading and commenting on an earlier draft of this paper. Dr. Pat Langley also provided a sounding board for ideas for extending research dealing with LTU trees.

References

Brent, R. P. 1990. Fast training algorithms for multi-layer neural nets. Numerical Analysis Project Manuscript NA-90-03, Dept. of Computer Science, Stanford Univ.

Brodley, C. E., and Utgoff, P. E. 1992. Multivariate Versus Univariate Decision Trees. COINS Technical Report 92-8, Dept. of Computer Science, Univ. of Mass.

Duda, R. O., and Hart, P. E. 1973. *Pattern Classification and Scene Analysis*. New York: John Wiley & Sons.

Langley, P. 1992. Induction of Recursive Bayesian Classifiers. Forthcoming.

Littlestone, N. 1988. Learning quickly when irrelevant attributes abound: a new linear-threshold algorithm. *Machine Learning* 2:285-318.

Littlestone, N. 1991. Redundant noisy attributes, attribute errors, and linear-threshold learning using Winnow. In Proceedings of the Fourth Annual Workshop of Computational Learning Theory, 147-156. San Mateo, CA: Morgan Kaufmann Publishers, Inc.

Nilsson, N. J. 1965. *Learning machines*. New York: McGraw-Hill.

Quinlan, J. R. 1986. Induction of decision trees. *Machine Learning 1*:81-106.

Ridgway, W. C., 1962. An Adaptive Logic System with Generalizing Properties. Stanford Electronics Laboratories Technical Report 1556-1, prepared under Air Force Contract AF 33(616)-7726, Stanford Univ.

Rumelhart, D. E.; Hinton, G. E.; and Williams, R. J. 1986. Learning internal representations by error propagation. *Parallel Distributed Processing, Vol. 1*, eds. D. E. Rumelhart and J. L. McClelland, 318-62. Cambridge, MA: MIT Press.

Rumelhart, D. E. and McClelland, J. L. eds. 1986. *Parallel Distributed Processing, Vol. 1.* Cambridge, MA: MIT Press.

Sahami, M. 1993. An Experimental Study of Learning Non-Linearly Separable Boolean Functions With Trees of Linear Threshold Units. Forthcoming.

Utgoff, P. E. 1988. Perceptron Trees: A Case Study in Hybrid Concept Representation. In AAAI-88 Proceedings of the Seventh National Conference on Artificial Intelligence, 601-6. San Mateo, CA: Morgan Kaufmann.

Widrow, B., and Winter, R. G. 1988. Neural Nets for Adaptive Filtering and Adaptive Pattern Recognition. *IEEE Computer, March*:25-39.

Winston, P. 1992. *Artificial Intelligence, third edition.* Reading, MA: Addison-Wesley.