To appear in *Proceedings of the First International IEEE Symposium on Intelligence in Neural and Biological Systems*, Washington, DC, May, 1995.

Generating Neural Networks Through the Induction of Threshold Logic Unit Trees

Mehran Sahami
Robotics Laboratory
Computer Science Department
Stanford University
Stanford, CA 94305

Abstract

This paper investigates the generation of neural networks through the induction of binary trees of threshold logic units (TLUs). Initially, we describe the framework for our tree construction algorithm and show how it helps to bridge the gap between pure connectionist (neural network) and symbolic (decision tree) paradigms. We also show how the trees of threshold units that we induce can be transformed into an isomorphic neural network topology. Several methods for learning the linear discriminant functions at each node of the tree structure are examined and shown to produce accuracy results that are comparable to classical information theoretic methods for constructing decision trees (which use single feature tests at each node), but produce trees that are smaller and thus easier to understand. Moreover, our results also show that it is possible to simultaneously learn both the topology and weight settings of a neural network simply using the training data set that we are initially given.

1 Introduction

We present a non-incremental algorithm that learns binary classification tasks by producing decision trees of threshold logic units (TLU trees). While similar to the decision trees produced by classical information theoretic algorithms such as ID3 [7], TLU trees seem to promise more generality and compactness of representation as each node implements a linear discriminant function as opposed to testing only one feature of the instance vector. Thus if the data to be classified does not align closely with the principle axes of the instance space, a large number of single feature tests may be required to properly separate the data set. On the other hand, using an *oblique* separating hyperplane may separate the data much more easily, thus requiring fewer separating hyperplanes to completely classify the data, and hence fewer nodes in the decision tree. This notion of using *multivariate* as opposed to *univariate* separating functions in the induction of decision trees has only very recently begun to attract the attention of researchers in the machine learning community [2, 3]. While earlier attempts in this direction have been made, most notably with the Perceptron Tree algorithm [14], even the Perceptron Tree does not capture the full generality of our structure. In Perceptron Trees only the leaf nodes of the tree may implement a general linear discriminant function. All internal nodes of the tree may only use univariate tests (similar to ID3) to shatter the instance space, again causing this algorithm to suffer from the same shortcomings as ID3. Recently, Brodley and Utgoff [2] have shown that learning multivariate (as opposed to univariate) decision trees also has the potential for greater generalization capabilities, but the results of these experiments are still preliminary and conflicting results have also been reported [12]. This report helps to address these issues as well.

Furthermore, we show how any such TLU tree can be mechanically transformed into a three-layer neural network as first suggested by Brent [1] and further developed by Sahami [10, 11]. Such transformed networks, which have two hidden layers and one output layer (the input layer is not counted) can often by trained much more quickly by building the TLU tree and transforming it into a network than attempting to train the corresponding network using the standard Back-Propagation algorithm [9] applied to the entire network.

In our investigation, we compare a number of different methods for learning the linear discriminant function at each node of the TLU tree [4, 9] and compare these with both the information theoretic approach to learning univariate tests (as in ID3) and a standard (naive Bayesian [6]) statistical approach to learning multivariate tests. We examine the performance of these algorithms on a wide variety of pattern classification and inductive learning tasks. Finally, we consider both experimental and

theoretical future work which still needs to be done in this newly emerging subfield of machine learning.

2 The TLU tree algorithm[1]

The tree building algorithm is non-incremental requiring that the set of all training instances, S, be available from the outset. We begin with the root node of the tree and induce a hyperplane to separate the training set 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 S_0 contains instances labeled 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 S_1 contains instances labeled 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.

3 Creating networks from TLU trees

The trees which are produced by the TLU tree algorithm can be mechanically transformed into three-layer connectionist networks that implement the same functions. Given an TLU 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 input features). The second hidden layer consists of *n* nodes (*AND* gates), where *n* is the number of possible distinct paths between the root of *T* and a fringe node (any node without two children). Finally, the output layer is merely a single *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 fringe node 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. An example is given below (Figures 1 and 2).

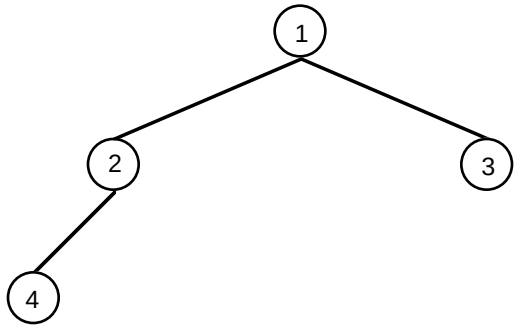

Figure 1

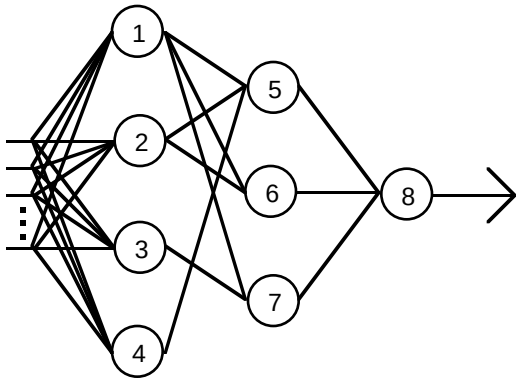

Figure 2

Figure 1 shows a decision tree produced by the TLU tree algorithm, and Figure 2 represents the corresponding network after performing the transformation described above. Nodes 1, 2, 3, and 4 in Figure 1 correspond directly to the same nodes in Figure 2. In Figure 2, 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 fringe node) 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. Node 8 is simply a disjunction (*OR* gate) of the outputs of nodes 5, 6 and 7.

[1]Notation in the description of the TLU tree algorithm is similar to that presented in [1]. The algorithm presented here closely follows [10].

4 Experimental results

4.1 Experimental considerations

In some of our TLU tree induction experiments, an error toleration factor was placed on the building of the trees in order to prevent needlessly complex trees — an application of Occam's razor — thereby helping to improve generalization. This toleration condition was set after some empirical observations which indicated that given some number of similarly classified instances in a node, a certain percentage of erroneous classifications, E, would be acceptable (thus precluding further branching for that particular classification from the node).

Additionally, we stopped partitioning along a branch if a node produced no better split of the instance space than its parent (while still making some errors). This served to prevent infinite recursion in constructing subtrees in which no better split than classifying all the instances into one class could be found.

In our experiments we compare a number of methods for partitioning the instance space at each node of the tree. We first implemented two classical methods for generating separating functions at each node of the TLU tree. The first method was our own variation of the ID3 algorithm in which single feature tests are selected based on minimizing entropy in the set of instances at each node. The entropy measure we minimized, as presented in [16], is given by:

$$\text{Entropy} = \sum_b \frac{n_b}{n_t} \cdot \left(\sum_c - \frac{n_{bc}}{n_b} \log_2 \frac{n_{bc}}{n_b} \right)$$

where n_t is the total number of instances in all branches, n_b is the number of instances in branch b, and n_{bc} is the total number of instances in branch b of class c. In our experiments $c \in \{0, 1\}$ since we consider only Boolean features.

The second classical method we implemented uses Bayesian statistical analysis to find an optimal separating hyperplane that minimizes the probability of error at each node of the tree assuming the components of the instance vector $\mathbf{X}$ are statistically independent. This type of discriminator is often called a *naive* Bayesian classifier. Simply stated, at each node we attempt to determine if

$$\frac{p(\text{class} = 1 \mid \mathbf{X})}{p(\text{class} = 0 \mid \mathbf{X})} \geq 1$$

in which case we would predict class 1 (right branch) for instance $\mathbf{X}$ and class 0 (left branch) otherwise.

Comparatively, we examine a number of adaptive techniques for inducing hyperplanes: the Perceptron training rule [6], the Least Mean Square (LMS) error algorithm [15], and Back-Propagation [9] applied to one neuron (also known as the Delta rule) to find separating hyperplanes. In applying the Perceptron error-correction rule to non-linearly separable data, we attempt to minimize the number of errors made at each node of the tree by using Gallant's Pocket algorithm [5] to keep the weight vector with the longest run of correct classifications at any point in our "pocket." The weight vector that remains in our pocket after we have finished training at a given node in the tree is then used as the linear separator at that node. Also, to allow the LMS algorithm to converge when presented with non-linearly separable data, we use a slowly decreasing learning rate parameter. As for Back-Propagation, since we are only allowed to store one hyperplane at each node (as opposed to an entire network, although this might be an interesting angle for further research) we apply the 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:

$$\text{Output}_{TLU} = \begin{cases} 1 & O_n \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

$$O_n = \frac{1}{1 + e^{-N}}, \text{ where } N = \mathbf{W} \bullet \mathbf{X}$$

where O_n is the actual real valued output of the nth trained unit on any instance and O_{TLU} is the output of the "linear threshold unit."

There are many possible extensions to the TLU tree induction algorithm, such as tree pruning in a post-processing phase [8], but such modifications are beyond the scope of this paper, and are only fine tunings to the underlying learning architecture.

4.2 Learning simple Boolean functions

The first set of experiments involve learning simple Boolean functions. These functions were especially chosen to capture the inherent characteristics of different distributions of vectors in the instance space. They include both linearly and non-linearly separable functions as well as conjunctive and k-of-n threshold concepts. Conjunctive concepts tend to be aligned with the principle axes of the instance space whereas k-of-n threshold concepts are not. The functions are given below.

Algorithm	3 bit-1 corner	3 bit-2 corners	5 bit-1 corner	5 bit-2 corners
Perceptron	100.00% ± 0.00%	92.50% ± 11.46%	100.00% ± 0.00%	91.88% ± 14.80%
Least Mean Sqr.	96.25% ± 5.73%	88.75% ± 10.38%	94.07% ± 3.26%	74.06% ± 9.79%
Back-Propagation	100.00% ± 0.00%	100.00% ± 0.00%	100.00% ± 0.00%	98.44% ± 2.88%
Naive Bayes	100.00% ± 0.00%	75.00% ± 0.00%	84.38% ± 0.00%	65.63% ± 0.00%
ID3	100.00% ± 0.00%	100.00% ± 0.00%	100.00% ± 0.00%	100.00% ± 0.00%

Table 1. Accuracy results on the simple boolean functions using 1,000 training instances at each node of the tree.

Algorithm	3 bit-1 corner	3 bit-2 corners	5 bit-1 corner	5 bit-2 corners
Perceptron	1.0 ± 0.0	2.2 ± 0.4	1.0 ± 0.0	6.9 ± 4.0
Least Mean Sqr.	1.8 ± 0.6	4.1 ± 1.4	8.7 ± 4.6	11.8 ± 7.2
Back-Propagation	1.0 ± 0.0	2.0 ± 0.0	1.0 ± 0.0	5.1 ± 2.4
Naive Bayes	1.0 ± 0.0	2.0 ± 0.0	3.0 ± 0.0	3.0 ± 0.0
ID3	3.0 ± 0.0	5.0 ± 0.0	14.0 ± 0.0	23.0 ± 0.0

Table 2. Number of nodes in the trees induced for the simple boolean functions using 1,000 training presentations.

3 bit - 1 corner

$\mathbf{X} \in \{0,1\}^3$ — class 1 if $\sum_{i = 1...3} x_i = 3$, else class 0

3 bit - 2 corners

$\mathbf{X} \in \{0,1\}^3$ — class 1 if $\sum_{i = 1...3} x_i = 0$ or 3, else class 0

5 bit - 1 corner

$\mathbf{X} \in \{0,1\}^5$ — class 1 if $\sum_{i = 1...5} x_i \leq 1$, else class 0

5 bit - 2 corners

$\mathbf{X} \in \{0,1\}^5$

class 1 if $\sum_{i = 1...5} x_i \leq 1$ or $\sum_{i = 1...5} x_i \geq 4$, else class 0

Our learning experiments involved two stages defined by the number of samples that each of the adaptive algorithms was presented with during training. For the Perceptron, LMS, and Back-Propagation algorithms, our initial trials allowed for 1,000 presentations of instances at each node of the tree during construction. In later experiments, this number was increased to 10,000 presentations. Instances were drawn randomly with replacement from a uniform distribution over the entire instance space, thus the learning algorithms were quite likely to see every instance in the instances space at least once. The naive Bayes and ID3 algorithms were provided with an exhaustive enumeration of all instances in the instance space (as these algorithms use statistical measures to find separating hyperplanes and thus are best served by simply having available a uniform distribution of samples over the entire instance space). The error toleration parameter, E, was set to 0% as there was no noise in the training instances. We averaged each algorithm over 10 runs, although the non-adaptive partitioning schemes will always induce the same decision tree given the same training data. The accuracy and standard deviation when tested on the entire instance space are reported in Table 1.

These initial results show that both the Perceptron and Back-Propagation algorithms are clearly effective learners within our TLU tree framework. The LMS algorithm seems to suffer in the case of learning non-linearly separable functions, but still outperforms the naive Bayesian algorithm, which does the worst overall. While ID3 has a very strong showing in terms of predictive accuracy, it requires far more hyperplanes to properly shatter the instance space for every problem than any of the adaptive multivariate methods. This results from the fact that the separators generated by ID3 must be parallel to the principle axes of the instance space. In all fairness to ID3 and the naive Bayesian classifier, they do learn more quickly than any of the adaptive approaches as they do not require repeated training on sample vectors, but only require accumulating statistics from the training set. Nevertheless, the adaptive methods still learn very rapidly — each run took from a few seconds to a few minutes.

The number of training presentations that the adaptive algorithms received at each node was then increased to 10,000. As shown in Tables 3 and 4, these methods clearly outperform ID3 and naive Bayes in accuracy and/or the compact size of the resulting trees. Note that the results for ID3 and naive Bayes are unchanged from above since these methods are not dependent on the number of sample presented if the entire instance space is available.

The results of our later experiments show that not only do the adaptive learning methods equal or surpass the accuracy of the non-adaptive methods for inducing decision trees, these trees are also easier for humans to understand

Algorithm	3 bit-1 corner	3 bit-2 corners	5 bit-1 corner	5 bit-2 corners
Perceptron	100.00% ± 0.00%	100.00% ± 0.00%	100.00% ± 0.00%	96.25% ± 11.25%
Least Mean Sqr.	96.25% ± 5.73%	92.50% ± 11.47%	96.88% ± 3.42%	97.19% ± 3.26%
Back-Propagation	100.00% ± 0.00%	100.00% ± 0.00%	100.00% ± 0.00%	100.00% ± 0.00%
Naive Bayes	100.00% ± 0.00%	75.00% ± 0.00%	84.38% ± 0.00%	65.63% ± 0.00%
ID3	100.00% ± 0.00%	100.00% ± 0.00%	100.00% ± 0.00%	100.00% ± 0.00%

Table 3. Accuracy results on the simple boolean functions using 10,000 training instances at each node of the tree.

Algorithm	3 bit-1 corner	3 bit-2 corners	5 bit-1 corner	5 bit-2 corners
Perceptron	1.0 ± 0.0	2.0 ± 0.0	1.0 ± 0.0	4.7 ± 1.7
Least Mean Sqr.	1.7 ± 0.6	2.4 ± 0.5	3.9 ± 1.5	12.2 ± 3.2
Back-Propagation	1.0 ± 0.0	2.0 ± 0.0	1.0 ± 0.0	2.0 ± 0.0
Naive Bayes	1.0 ± 0.0	2.0 ± 0.0	3.0 ± 0.0	3.0 ± 0.0
ID3	3.0 ± 0.0	5.0 ± 0.0	14.0 ± 0.0	23.0 ± 0.0

Table 4. Number of nodes in the trees induced for the simple boolean functions using 10,000 training presentations.

given their small size. The trees generated by Back-Propagation are not only perfectly accurate, but they also reflect the minimum number of hyperplanes required to separate the instance space. Thus the TLU tree algorithm provides a means to gauge the "difficulty" of these functions as reflected in the size of the generated trees. Thus it is possible to simultaneously learn both the topology of a network needed to learn a function (as reflected by transforming the resultant tree into a network) and to learn the function itself. Moreover, we can use the TLU tree algorithm as an initial step in neural network design to give us an idea as to how large a network should be to learn a given function. We can first generate the appropriate TLU tree, transform it into a network with "informed" initial weights and then continue to train the network using a network training method.

4.3 The Monks problems

While we have shown that TLU tree generation is successful in simple Boolean learning tasks, it is more important to examine the performance of the our TLU tree algorithms on more standard learning tasks (in which the training set only includes a small fraction of the entire instance space). For these experiments, we the Monk's Problems [13]. The original description of these problems is given below.

Each vector **X** has 6 attributes, where:

attribute 1 (a1) ∈ {1, 2, 3}
attribute 2 (a2) ∈ {1, 2, 3}
attribute 3 (a3) ∈ {1, 2}
attribute 4 (a4) ∈ {1, 2, 3}
attribute 5 (a5) ∈ {1, 2, 3, 4}
attribute 6 (a6) ∈ {1, 2}

Each vectors is classified as 0 or 1.

Monk's Problem 1 (Monk 1)
if (a1 = a2) or (a5 = 1) then 1, else 0
No noise in training set.
124 instances out of 432 in training set.

Monk's Problem 2 (Monk 2)
if *exactly* two of the 6 attributes = 1 then 1, else 0
No noise in training set.
169 instances out of 432 in training set.

Monk's Problem 3 (Monk 3)
if (a5 = 3 and a4 = 1) or (a5 ≠ 4 and a2 ≠ 3) then 1, else 0
5% class noise in training set.
122 instances out of 432 in training set.

This set of three learning tasks has been well studied on a number of different machine learning algorithms and includes standard training and test sets to help make fair comparisons between methods. While the standard Monk's Problems use nominal instance vectors in a 6-dimensional space, we encoded these vectors into a 17-dimensional Boolean space using local encoding.

In our experiments, we used 10,000 samples drawn randomly with replacement from the training set to train the nodes of our TLU trees for the adaptive learning methods. The statistical learning methods had available the entire training set from which to compute frequency statistics. Since two of the Monks Problems are noise free and one contains noise, we ran the TLU tree induction algorithm using an error toleration parameter, E, of 0% and 10% in two different sets of experiments. We averaged the algorithms over 10 runs. The results of the first set of experiments (with E = 0%) are shown in Tables 5 and 6, with the induced TLU trees being tested on the entire instance space.

Algorithm	Monk 1	Monk 2	Monk 3
Perceptron	83.87% ± 6.26%	75.53% ± 5.27%	92.94% ± 0.98%
Least Mean Sqr.	92.22% ± 3.47%	76.92% ± 6.49%	92.71% ± 2.18%
Back-Propagation	100.00% ± 0.00%	93.98% ± 3.72%	93.98% ± 0.80%
Naive Bayes	70.83% ± 0.00%	67.13% ± 0.00%	97.22% ± 0.00%
ID3	92.59% ± 0.00%	86.57% ± 0.00%	89.81% ± 0.00%

Table 5. Results on the Monks Problems learning tasks with E = 0%.

Algorithm	Monk 1	Monk 2	Monk 3
Perceptron	11.0 ± 2.8	23.9 ± 7.4	4.8 ± 0.9
Least Mean Sqr.	13.3 ± 1.8	42.2 ± 8.9	12.1 ± 2.0
Back-Propagation	3.0 ± 0.0	7.4 ± 2.8	3.7 ± 0.8
Naive Bayes	5.0 ± 0.0	2.0 ± 0.0	5.0 ± 0.0
ID3	20.0 ± 0.0	45.0 ± 0.0	18.0 ± 0.0

Table 6. Number of nodes in the trees induced in the Monks Problems learning tasks with E = 0%.

Algorithm	Monk 1	Monk 2	Monk 3
Perceptron	82.27% ± 4.55%	73.52% ± 3.98%	95.92% ± 1.55%
Least Mean Sqr.	88.01% ± 3.88%	77.13% ± 2.44%	94.21% ± 2.51%
Back-Propagation	91.67% ± 0.00%	91.18% ± 3.35%	94.45% ± 0.99%
Naive Bayes	70.83% ± 0.00%	67.13% ± 0.00%	97.22% ± 0.00%
ID3	92.59% ± 0.00%	85.19% ± 0.00%	97.22% ± 0.00%

Table 7. Results on the Monks Problems learning tasks with E = 10%.

Algorithm	Monk 1	Monk 2	Monk 3
Perceptron	6.3 ± 2.5	21.8 ± 6.0	1.7 ± 0.9
Least Mean Sqr.	10.8 ± 4.7	38.8 ± 7.0	5.0 ± 3.2
Back-Propagation	2.0 ± 0.0	6.1 ± 2.8	1.1 ± 0.3
Naive Bayes	5.0 ± 0.0	2.0 ± 0.0	1.0 ± 0.0
ID3	20.0 ± 0.0	37.0 ± 0.0	2.0 ± 0.0

Table 8. Number of nodes in the trees induced in the Monks Problems learning tasks with E = 10%.

The TLU trees produced using Back-Propagation seem to be clearly outperform the non-adaptive methods when considering a combination of accuracy and tree size. While the trees produced by the naive Bayesian classifier on Monk 2 are smaller than those produced using Back-Propagation, the accuracy results are far inferior. Interestingly, the trees induced using Perceptron and LMS are not nearly as small as we expected them to be, and their accuracy also seems to suffer as a result of not being able to learn a parsimonious multivariate shattering of the instance space. We conjecture that this problem results from the algorithms' inability to converge when presented with high-dimensional, non-linearly separable data. Nevertheless, Back-Propagation seems to provide a very effective method of learning TLU trees, producing smaller and more accurate trees than Perceptron, LMS, and ID3 in every problem.

Tables 7 and 8 show the results of running the tree induction algorithms using an error toleration parameter of 10%. We hypothesize that this error toleration parameter would induce smaller trees with some (small) loss of accuracy for the noiseless problems (Monk 1 and 2) and actually increase accuracy by reducing overfitting in the problem that contains noise (Monk 3).

The experimental results confirm our hypotheses as every algorithm generally produces smaller trees for all three learning tasks and the accuracy of virtually all of the algorithms is improved with respect to Monk 3. ID3 and Back-Propagation appear to be the most accurate algorithms. ID3 slightly edges Back-Propagation in accuracy on Monk 1 and 3, since the concept to be learned is closely aligned with the principle axes of the instance space lending itself well to the inherent bias in ID3. However, in Monk 2 Back-Propagation outperforms ID3 as this concept is skewed in relation to the principle axes of the instance space and thus lends itself to an oblique partitioning. Nevertheless, ID3 still outperforms Perceptron and LMS on Monk 2, again supporting our conjecture that these particular algorithms (even with the modifications previously described to help them converge) are not robust in the presence of high dimensional non-linearly separable data.

Algorithm	*E=0%* Accuracy	Node Count	*E=10%* Accuracy	Node Count
Perceptron	59.76% ± 19.56%	12.4 ± 3.4	68.81% ± 15.70%	9.8 ± 2.2
Back-Prop	69.76% ± 27.07%	10.1 ± 1.2	79.29% ± 20.34%	8.6 ± 3.5
Naive Bayes	68.10% ± 20.01%	5.8 ± 1.1	92.38% ± 9.98%	5.3 ± 1.6
ID3	89.29% ± 12.11%	18.5 ± 1.7	96.90% ± 6.21%	18.5 ± 1.7

Table 9. Accuracy results and node counts in the trees induced on the Mux6 learning task.

Algorithm	*E=0%* Accuracy	Node Count	*E=10%* Accuracy	Node Count
Perceptron	94.96% ± 3.02%	3.5 ± 1.0	95.86% ± 2.47%	1.0 ± 0.0
Back-Prop	96.09% ± 2.07%	4.1 ± 0.9	96.09% ± 2.08%	1.0 ± 0.0
Naive Bayes	91.72% ± 2.35%	6.0 ± 0.0	91.72% ± 2.35%	2.0 ± 0.0
ID3	94.48% ± 2.10%	23.1 ± 2.3	95.64% ± 1.89%	1.0 ± 0.0

Table 10. Accuracy results and node counts in the trees induced on the Vote learning task.

In terms of tree size, the trees induced using Back-Propagation give the most parsimonious representations while still maintaining good classification accuracy. This is most clearly seen in Monk 2 where the Back-Propagation TLU trees are far more accurate than ID3 while producing trees that have 5 times fewer nodes. Also, in Monk 1 Back-Propagation sacrifices less than one percentage point of accuracy while producing a tree that is an order of magnitude smaller than that produced by ID3. The results from Monk 3 are inconclusive as a single partition produces greater than 90% accuracy (within the 10% error toleration parameter) for virtually every learning algorithm.

4.4 Real-world data sets

The strengths and weaknesses of the adaptive vs. non-adaptive methods are most clearly seen when the TLU tree algorithm is run on two real-world learning tasks with quite different characteristics. The first task is predicting the output of a multiplexer with 2 address bits and 4 input bits (*Mux6*). Thus the instance vectors are in $\{0, 1\}^6$ and the class is simply 0 or 1. The entire space of 64 instance vectors was used in our study. The second learning task involved predicting the political party (Democrat or Republican, encoded as 1 or 0) of 435 Congressmen based on their votes on 16 key issues (*Vote*). Each attribute could thus take on the value "yea", "nay", or unknown. These three-valued features were locally encoded into a Boolean feature space. We used 10-fold cross-validation in our experiments along with error toleration values of 0% and 10%. Note that since the Least Mean Square method seemed to fare so poorly in our earlier experiements, it was excluded from this one. The results for the Mux6 and Vote domains are shown in Tables 9 and 10 respectively.

In the Mux6 domain, the non-adaptive methods clearly outperform the adaptive ones. The standard deviations in this experiment are so large, however, that we can not be overy confident in these results. Nevertheless, it seems that in some situations single bit tests (as performed by ID3) can produce favorable results when the problem to be learned is aligned well with the axes of the instance space. In such cases, the inherent bias in algorithms such as ID3 play a crucial role in the generalization ability of the learning algorithm. The extra degrees of freedom in the multivariate methods allow them to overfit the data and thus generalize very poorly. This is also clear from the fact that when the error toleration paramenter is increased to 10%, the multivariate methods fare much better relatively, as they are no longer overfitting the training data as much.

In the Vote domain, the adaptive methods emerge as the most accurate while producing the smallest decision trees for both settings of the error toleration parameter. ID3 produces relatively acurate, but very large, trees when E is 0%. Moreover, ID3 is never quite able to attain the same level of accuracy as that achieved with Perceptron or Back-Propagtion, but this difference is rather small and not statistically significant. Interestingly, it is possible to perform very well in this domain with only one linear separation, thus arguing against having a complex neural network topolgy. This is an important finding when considering how large a hand-crafted network should be when used to learn a given domain.

5 Conclusions

There are several conclusions based on this work which have been outlined above. Foremost is the ability to actually learn neural network topologies through the construction of TLU trees using Back-Propagation. Not only can the TLU tree algorithm be used as a stand-alone learning algorithm, but it can also be used as a starting point to get a rough approximation of the size of network necessary to properly learn a given data set. Moreover, by inducing a TLU tree, we can produce an initial set of

weights for a network which we can further train incrementally as new training data becomes available. Thus this learning architecture helps bridge the gap between entirely incremental and entirely non-incremental learning methods.

Furthermore, we have seen that in both simple Boolean and complex learning tasks, the TLU tree is a viable stand-alone learning algorithm that performs just as well (if not better) than the widely used information theoretic and statistical approaches to inducing classification trees and often produces much more compact representations. However, non-robust methods such as Perceptron and LMS are not necessarily resilient to high-dimensional, non-linearly separable data. We must also be cautious in some cases to not use a multivariate method if the domain to be learned does align well with the principle axes of the instance space.

6 Future Work

There is still a great deal of work, both theoretical and experimental, which needs to be conducted in the study of TLU trees. There are many methods for learning separating hyperplanes which have not yet been explored within the framework of inducing TLU trees. To promote generality, methods for TLU tree pruning must be examined. Another interesting angle for further work involves growing actual neural networks which are isomorphic to decision trees rather than growing the trees themselves. In this way, we can effectively learn all the nodes of the tree simultaneously and add nodes to the network when it appears we have reached a local minimum in training. Finally, there are a number of theoretical questions in regards to TLU trees that need to be answered, such as the capacity of these trees, or how minimizing error locally in each node may effect the number of nodes needed to minimize error globally.

Acknowledgments. We are indebted to Nils Nilsson for providing the initial guidance and support to pursue this line of research. Discussions with George John and Pat Langley have also provided helpful insights regarding the induction of decision trees and pruning methods. The author is supported by a Fred Gellert Foundation ARCS fellowship.

References

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

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

[3] Brodley, C. E., and Utgoff, P. E. 1994. Multivariate Decision Trees. To appear in *Machine Learning*.

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

[5] Gallant, S. I. 1986. Optimal Linear Discriminants. In *Eighth International Conference on Pattern Recognition*, 849-852. New York: IEEE.

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

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

[8] Quinlan, J. R. 1993. *C4.5: Programs For Machine Learning*. San Mateo, CA: Morgan Kaufmann.

[9] 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.

[10] Sahami, M. 1993. Learning Non-Linearly Separable Boolean Functions With Linear Threshold Unit Trees and Madaline-Style Networks. In *AAAI-93 Proceedings of the Eleventh National Conference on Artificial Intelligence*, 335-41. Menlo Park, CA: AAAI Press.

[11] Sahami, M. 1995. Generating Neural Networks Through the Induction of Threshold Logic Unit Trees (Extended Abstract). To appear in *ECML-95 Proceedings of the Eighth European Conference on Machine Learning.*

[12] Schaffer, C. 1994. Linear Discriminant Tree Induction Algorithms Compared (Extended Abstract). Working paper.

[13] Thrun, S. B., and 23 co-authors. 1991. The monk's problems: a performance comparison of different learning algorithms. Technical Report CMU-CS-91-197, Carnegie Mellon University.

[14] 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.

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

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