Perceptron Trees: A Case Study in Hybrid Concept Representations

PAUL E. UTGOFF

This article presents a case study in examining the bias of two particular formalisms: decision trees and linear threshold units. The immediate result is a new hybrid representation, called a 'perceptron tree', and an associated learning algorithm called the 'percepton tree error correction procedure'. The longer term result is a model for exploring issues related to understanding representational bias and constructing other useful hybrid representations.

1. Introduction

One of the open problems in machine learning is how to learn from examples, also known as supervised learning. One would like to observe instances whose class memberships are known, and identify generalizations that are correct for these observed instances and for others whose class memberships are not known. Many algorithms have been devised, including candidate elimination (Mitchell, 1978), AQ (Michalski & Chilausky, 1980), ID3 (Quinlan, 1983, 1986), and back propagation (Rumelhart & McClelland, 1986).

A fundamental issue in concept learning is the problem of built-in biases that cause some generalizations to be preferred to others, even among those generalizations that are consistent with all the observed training instances (Utgoff, 1986). One source of bias is the representation, which consists of two components. The first is the legal syntax or form of concept descriptions, here called the *formalism.* Examples include predicate calculus, formal grammars, set-theoretic notation, and connectionist networks. The second component is the set of terms that provide the basic building blocks for constructing concept descriptions within the given formalism.

This article is concerned with biases that are inherent in a given representational formalism. A space of concept descriptions that is easy to define in one formalism may be difficult to define in another. This impacts the order in which a learning algorithm searches the space and therefore how it chooses its current hypothesis of the concept to be learned.

It is beyond the scope of this article, and beyond our present knowledge, to make any catalogue of representational formalisms and their inherent biases, or to draw any large conclusions about such biases. Instead, the article presents a case study in examining the inherent bias of two particular formalisms: decision trees and linear

Paul E. Utgoff, Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003, USA. Tel: (413) 545-4843; Net: utgoff@cs.umass.edu.

threshold units. The immediate result is a new hybrid representation, and an associated learning algorithm. The longer term result is a model for exploring issues related to understanding representational bias and constructing other useful hybrid representations.

2. Motivation

The thesis of the work is that individual concept formalisms have inherent biases, which implies that no one of them is the best choice for all concept learning problems. It would increase the autonomy and effectiveness of a learning program if it were able to select and mix formalisms, allowing a *hybrid representation.* One would like to be able to draw on the special strengths of the available formalisms. *Strength* is used loosely to refer to the ease with which particular concepts can be learned within the formalism. To the extent that the strength of each individual representation complements the weaknesses of the others, the hybrid representation is enriched.

The present work arose from the need for a learning program to be able to handle a stream of training instances flowing at a rate of up to several thousand instances per minute (Utgoff & Heitman, 1988). This comes about from inferring pairs of states, during problem solving, in which one state is preferred to the other because it is on a path to a best solution and the other is not. Such pairs can be generated rapidly and in great abundance. One would like an incremental learning method capable of processing each training instance rapidly. In terms of handling a large volume of instances, decision tree methods and connectionist learning methods are natural choices.

The desire to find concepts that are consistent with *all* the training instances, given that the training instances are labeled consistently favored decision trees, leading to examination of ID3 (Quinlan, 1983), ID4 (Schlimmer & Fisher, 1986), and ID5R (Utgoff, 1989). Unfortunately, ID3 is not incremental, ID4 does not always find a consistent concept description, and ID5R saves selected training instances, making it potentially space-inefficient for a large volume of instances. More important, the bias of the decision tree formalism is inappropriate for many kinds of concepts that one would like to be able to learn.

Connectionist methods provide the needed efficiency in handling individual training instances, but there is no existing theory regarding choice of network architecture. This is of critical importance, because choice of network is closely analogous to the problem of selecting a representation. For example, a large network can represent more concepts than a small network. The choice of network architecture directly affects the expense of updating weights, granularity of representation, and quality of generalization (Hinton, 1989).

The following requirements and assumptions emerged:

(1) The learning algorithm must be able to handle a large volume of training instances efficiently and incrementally.
(2) The algorithm must be able to select an appropriate formalism at any level.
(3) The algorithm must find a consistent concept description in finite time without human intervention.
(4) The training instances are assumed to be labeled consistently. This means that any given training instance is labeled with the same class name on every presentation.
(5) The resulting concept description must be efficient for classifying unobserved instances.

Now compare the characteristics of learning a decision tree to those of learning a linear threshold unit.

3. Decision Trees and Linear Threshold Units

A *decision tree* is a node that contains a class name or a test with, for each possible outcome of the test, a branch to a decision tree. The set of available branches at a node represents a disjunction. Each distinct path through the tree, from the root to an answer node, represents a conjunction. For classification purposes, a decision tree is traversed, starting at the root, according to the decision nodes in the tree and the corresponding values in the instance, until an answer node is reached. An answer node contains the name of the class to which the instance is inferred to belong. There is a large literature on methods for constructing decision trees; for a survey, see Moret (1982). Throughout this paper, the information-theoretic approach is assumed (Lewis, 1962; Quinlan, 1983), in which the tree building process selects an attribute test that removes the greatest amount of ambiguity, leaving the least amount of expected decision making to be done. This kind of tree building procedure is not optimal, but it does generally find small trees. The structure of a decision tree has the effect of partitioning the instance space at each decision node, due to the manner in which the tree is traversed for classification purposes. A subtree is then built recursively, as necessary, for each subspace of instances in the partition.

A *linear threshold unit*, herein abbreviated LTU, is a device that compares a weighted sum of instance features to a threshold value (Minsky & Papert, 1972). The model assumes that presence or absence of a feature in an instance is represented numerically. An instance is represented by a vector $\mathbf{I}$ that encodes the presence or absence of each feature. The LTU maintains one weight for each feature and one weight for the threshold, making a vector $\mathbf{W}$ of such weights. The threshold, denoted θ, is treated simply as a feature that is present in every training instance. If $\mathbf{W}\cdot\mathbf{I}$ is greater than 0 then the instance $\mathbf{I}$ is classified as a positive instance by the LTU. If $\mathbf{W}\cdot\mathbf{I}$ is less than 0 then the instance $\mathbf{I}$ is classified as a negative instance. Otherwise, the classification is unknown. Here, 'positive' and 'negative' are the two standard class names for the examples and counterexamples of a single concept. Geometrically, $\mathbf{W}$ defines a hyperplane. The inner product of $\mathbf{W}$ and $\mathbf{I}$ indicates which side of the hyperplane $\mathbf{I}$ is on. As guaranteed by the perceptron convergence theorem (Minsky & Papert, 1972), a $\mathbf{W}$ that separates the positive and negative instances via a hyperplane can be found in a finite number of steps if such a $\mathbf{W}$ exists. This means that a LTU will find a consistent concept description if and only if the target concept is describable by a hyperplane that separates the positive and negative instances.

Now consider the characteristics of the two formalisms and their associated learning algorithms as listed in Table I. The item 'complete representation' refers to whether every concept over the instance space is representable. 'Guaranteed convergence' indicates whether the associated learning algorithm is guaranteed to find a concept description that is consistent with all the observed training instances. 'Efficient update' refers to the expense of handling a training instance that has been presented to the learning algorithm. The expense of classifying an instance is different for the two methods, as indicated by 'efficient classifier' and 'features evaluated'. For a decision tree, features are evaluated one at a time as the decision tree is traversed to a leaf node. Features that are not needed for determining the classification are not evaluated. This is desirable for some applications in which evaluating variables is costly or has side effects,

such as medical diagnosis. One does not want to conduct unnecessary tests. For a LTU, all features must be evaluated in order to determine the classification.

Table I. Characteristics of the two methods

	Decision tree	LTU
Complete representation	Yes	No
Guaranteed convergence	Yes	No
Efficient update	No	Yes
Efficient classifier	Yes (very)	Yes
Features evaluated	Some	All
Boolean combination bias	Yes	No
Hyperplane bias	No	Yes

In qualitative terms, one would like a representation and associated learning algorithm that possesses all the favorable and none of the unfavorable characteristics. Note that neither decision trees nor LTUs alone possess all the favorable characteristics, but that collectively they do.

4. Perceptron Trees

This section reports a case study in constructing a hybrid representation and associated learning algorithm. It is motivated by the requirements listed above in Section 2 and by the observation in Table I that decision trees and linear threshold units complement each other well. The desired characteristics are available in one or the other of the formalisms.

4.1. Perceptron Tree Representation

Define a *perceptron tree* to be either a linear threshold unit, or an attribute test with, for each value the attribute can take on, a branch to a perceptron tree. The term 'perceptron tree' is chosen because the linear threshold unit is the basic unit of Rosenblatt's perceptron. A perceptron tree is much like a decision tree, except that every leaf node is a LTU. As explained below, this is not simply a case of trading in answer nodes for LTUs. Given the ability of a LTU to represent concepts, a LTU can serve in place of a decision tree or subtree. The number of decision nodes in a perceptron tree need never exceed the number of nodes in a plain decision tree, and will typically be fewer.

For the work reported here, the *symmetric* model of instance representation is assumed (Hampson & Volper, 1986). Each feature is represented by 1 if present in the instance and -1 otherwise. A *feature* is a specific value of a specific attribute. For example, if the color of the instance is red, then the attribute is 'color', the value is 'red', and the feature is 'color is red'. Thus, all attributes are assumed to be discrete-valued.

It is important to note that the perceptron tree formalism is complete.

Theorem 1. The perceptron tree formalism is complete, in the sense that for every possible subset of the instance space there is a perceptron tree that can describe exactly that subset.

Proof. The decision tree formalism is complete because DNF is complete. Because a perceptron tree could be elaborated to a plain decision tree, to the point that each instance is described by a single attribute, it is sufficient to show that a LTU can discriminate instances described by a single feature. This is trivially so because for each attribute value i observed in some positive instance, some weight $w_i > 0$ will cause that instance to be classified positive. Similarly, for each attribute value i observed in some negative instance, some weight $w_i < 0$ will cause that instance to be classified negative. Under the assumption that a training instance is never labeled positive on one occasion and negative on another, it will always be the case that an instance with a given value of the attribute can be uniquely classified.

4.2. Perceptron Tree Error Correction Procedure

The error correction procedure incrementally updates a perceptron tree, which is a global data structure. The initial perceptron tree consists of a single empty node. An *empty node* is a node that contains no information and has not yet been initialized as either as a decision node or a LTU node. A *decision node* is a node that contains an attribute test and, for each value of the test attribute that has been observed previously, a branch to a perceptron tree. A *LTU node* is a node that contains a linear threshold unit. The notation dim(**W**) indicates the number of components (features) in vector **W**. Table II shows the procedure that updates a perceptron tree in response to a training instance, called the *perceptron tree error correction procedure.*

Table II. The perceptron tree error correction procedure

1. While the root node of the perceptron tree or subtree is a decision node, and there exists a value branch corresponding to the value in the instance of the attribute tested, traverse the value branch to its subtree.
2. If the root of the perceptron tree or subtree is a decision node, then there was not value branch to traverse in step 1. Add a new branch with a new empty node at its leaf and traverse the new branch to its subtree.
3. If the root of the perceptron tree or subtree is an empty node, then make it a LTU node and initialize the LTU at the node. The LTU is initialized by setting all weights in the vector **W** of the LTU to zero. Any other bookkeeping variables for the LTU are also initialized.
4. At this point, the perceptron tree or subtree is guaranteed to be a LTU node. Compute the relationship of instance **I** to the hyperplane defined by **W** by $y \leftarrow \mathbf{W} \cdot \mathbf{I}$.
5. If $y > 0$ and the training instance is negative, then adjust **W** so that **I** would have been classified correctly as negative. This is computed by

$$\mathbf{W} \underset{\sim}{\leftarrow} \mathbf{W} - \mathbf{I}\left(\left\lfloor \frac{y}{\dim(\mathbf{W})} \right\rfloor + 1\right).$$ Go to step 7.

6. If $y < 0$ and the training instance is positive, then adjust **W** so that **I** would have been classified correctly as positive. This is computed by $\mathbf{W} \leftarrow$

$$\mathbf{W} \underset{\sim}{\leftarrow} \mathbf{W} - \mathbf{I}\left(\left\lceil \frac{y}{\dim(\mathbf{W})} \right\rceil - 1\right).$$

7. If the space of instances at this node should be partitioned into subspaces, then discard the LTU at this node and replace it with an attribute test. This makes the node a decision node with no branches. (There is no immediate need to provide branches below the node because they will be grown as necessary with subsequent training.)

There are four points to note. First, the procedure indicated for adjusting **W** in steps 5 and 6 above is a special case of the *absolute error correction procedure* described in Nilsson (1965). Second, **W** is integer-valued, though any fixed resolution will suffice. Third, a perceptron tree only grows, it never shrinks. Finally, the **W** at each LTU corresponds to the features that were not determined by decision nodes. For example, if 'color' is a test attribute above a given LTU, then no feature with attribute 'color' is part of the **W** of that LTU. This is because the attribute 'color' and its value are fixed as a result of taking that path through the decision nodes of the perceptron tree. Thus, each subproblem is of lower dimensionality.

There are two issues in step 7 of the error correction procedure. First is the problem of detecting when the space of instances should be partitioned via an attribute test. The second is the problem of picking the attribute for the decision node of the perceptron tree. A specific method for deciding when to partition, and a specific method for picking an attribute are given below. Together, they illustrate one way of instantiating step 7 of the procedure. The sole requirement is that the space of instances at a node be split if that space is not linearly separable.

4.2.1. When to split. If the space of instances at a node is not linearly separable, then it is necessary that the space be split (partitioned) into subspaces. A space of instances is *linearly separable* if there exists a hyperplane that discriminates the positive and negative training instances. The problem is to detect that the space of instances is not linearly separable. The Perceptron Cycling Theorem (Minsky & Papert, 1972) states that the perceptron learning algorithm visits a finite number of weight vectors **W** regardless of separability. A corollary (Gallant, 1986) is that the perceptron learning algorithm will leave and revisit at least one weight vector if and only if the space of instances is not linearly separable. Thus, to prove nonlinear separability, it is sufficient to prove that the current weight vector **W** has been visted before. A sufficient test for separability is:

Corollary 1. If the number of vectors visited (so far) exceeds the number of distinct vectors that could have been visited (so far), then the space of instances is not linearly separable.

To be able to compute an upper bound on the number of distinct vectors that could have been visited so far, the minimum and maximum value that each weight w_i has ever taken on are maintained within the LTU. The notations $w_{i,\min}$ and $w_{i,\max}$ indicate, respectively, the minimum and maximum values w_i has ever taken on. Assuming integer-valued weights, an upper bound on the number of distinct vectors that could have been visited is:

$$\prod_{i=1}^{\dim(\mathbf{W})} (w_{i,\max} - w_{i,\min} + 1) \qquad (1)$$

This leads immediately to:

Corollary 2. Nonlinear separability can be detected in a finite number of steps, without saving previous weight vectors.

This follows immediately because the above upper bound on the number of distinct weight vectors that could have been visited is finite. Thus, by corollary 1 and the above computable upper bound (1) on the number of distinct vectors that could have

been visited so far, a procedure exists for detecting nonlinear separability: if the number of vectors visited (so far) exceeds upper bound (1), then the space of instances is not linearly separable because at least one weight vector will have been visited twice.

This test for nonlinear separability is correct, but conservative because the upper bound is not tight. Cycling can occur long before the test detects it. A test is needed that both detects cycling when it first occurs and does not require saving the training instances. Ho & Kashyap (1965) constructed a procedure that searches for an assignment of function values to each of the training instances such that the weight vector **W** is a solution to the resulting set of linear equalities. The procedure detects when such an assignment is impossible, indicating that the space of instances is not linearly separable. Unfortunately, the procedure is not amenable to incremental learning. Short of saving all versions of the weight vector **W** and checking for duplication, no other procedure for detecting nonseparability is known.

Lacking a practical method for determining nonseparability, one must employ a more aggressive heuristic test for deciding when to split. Due to the completeness of the perceptron tree representation, splitting more often than is strictly necessary is not harmful, in the sense that the ability to find a consistent concept description is not lost. It only means that it is possible that a decision node will have split the space even though a LTU would have been sufficient. Instead of waiting for a guarantee that the space of instances is not linearly separable, one would like to detect when the LTU is not making significant progress toward arriving at a consistent concept description.

The specific test used in the implementation described below is based on the number of vector adjustments of **W** that have occurred since some $w_{i,\,\min}$ or some $w_{i,\,\max}$ has been adjusted. If **W** continues to be adjusted in response to misclassified training instances, yet the historical minimum and maximum values of the w_i come to be adjusted rarely or seemingly not at all, then there is reason to believe that there is lack of progress in moving toward a solution vector. At issue is how many weight adjustments without changing a $w_{i,\,\max}$ or a $w_{i,\,\min}$ constitute lack of progress. Let C be the number of consecutive vector adjustments to **W** since some $w_{i,\,\min}$ or some $w_{i,\,\max}$ has been adjusted. The test in the current implementation of the algorithm is: if $C > \log \dim(\mathbf{W})$ then split the space of instances. This particular test was determined empirically, and is motivated by the observation that a maximum or minimum for some w_i will be changed often while **W** is making good progress toward a solution vector.

Duda & Hart (1973) point out that the magnitude of the weight vector **W** grows relatively quickly during initial training and then vascillates within a limited range. Experiments using a test based on detecting such vascillation generally led to more aggressive splitting than the test given above. Apparently, the magnitude of the vector stabilizes before its orientation. The test given above is sensitive to the orientation of the weight vector.

An alternative test could be constructed in terms of Gallant's (1986) observation that an optimal linear discriminant will achieve the longest run of consecutive correct classifications. His Pocket Algorithm calls for retaining 'in one's pocket' the **W** that has had the longest run so far. Gallant points out that a test could be devised that is sensitive to how often a better linear discriminant is found. Experiments with such a test are currently underway.

4.2.2. Where to split. The problem of picking an attribute test for a decision node has received much attention in the fields of pattern recognition and statistics (Fu, 1968;

Moret, 1982). As mentioned above in Section 3, the approach taken here is to employ an information-theoretic criterion, based on entropy (Shannon, 1948; Lewis, 1962), that measures the amount of ambiguity in a space of instances. The attribute that removes the greatest amount of ambiguity, by partitioning the space into the least ambiguous subsets, is chosen as the attribute test for the decision node.

As per Quinlan (1986), ambiguity of a partition of the instance space on a test attribute is measured by the function E(attribute) given below. The attribute with the lowest E-score is assumed to give a good partition of the instances into subproblems for classification. See section 3.3.1 of Moret (1982) for a general discussion of splitting criteria. See Mingers (1989) for an empirical comparison of various splitting criteria.

In order to describe the splitting criterion, the set of attributes used to describe an instance is denoted by A, and the individual attributes are indicated as a_i, with i between 1 and the total number of attributes, denoted $|A|$. For each attribute a_i, the set of possible values for that attribute is denoted V_i. The individual values are indicated by v_{ij}, with j between 1 and the total number of values for attribute a_i, denoted $|V_i|$. At a given node, let p=number of positive instances; n=number of negative instances; p_{ij}=number of positive instances with value v_{ij} of attribute a_i, and n_{ij}=number of negative instances with value v_{ij} of attribute a_i. Then

$$E(a_i)=\sum_{j=1}^{|V_i|}\frac{p_{ij}+n_{ij}}{p+n}\,I(p_{ij},n_{ij})$$

with

$$I(x,y)=\begin{cases}0 & \text{if } x=0\\ 0 & \text{if } y=0\\ -\dfrac{x}{x+y}\log\dfrac{x}{x+y}-\dfrac{y}{x+y}\log\dfrac{y}{x+y} & \text{otherwise}\end{cases}$$

The above information-theoretic splitting criterion requires knowing the number of positive and number of negative instances observed for each of the w_i. These counts are maintained in each LTU, and are updated for every observed training instance, whether or not the weights in **W** are adjusted. Thus, while a LTU is being trained, information is also maintained that would make it possible to replace the LTU with a decision node.

4.2.3. Convergence to a consistent concept description. Given that there exists a perceptron tree representation of a concept description that is consistent with all the training instances, one needs to consider whether such a description will be found.

Theorem 2. If the training instances are labeled consistently, then the perceptron learning algorithm, using the perceptron tree error correction procedure, will find a consistent concept description in a finite number of steps.

Proof. Either the LTU finds a solution vector in a finite number of steps, as per the Perceptron Convergence Theorem, or the space of instances is detected to be not linearly separable in a finite number of steps (corollary 2). If the space of instances is

not linearly separable, then it is split with an attribute test. Since the algorithm is applied recursively at each node, it is only necessary to show that a linearly separable space is finally reached at each LTU node. This is guaranteed by the completeness of the representation (theorem 1) and the consistent labeling assumption.

4.3. An Illustration

To illustrate various characteristics of learning with the perceptron tree error correction procedure, a simple problem is given here. The problem is to learn the concept

$$(a \vee b) \oplus (c \wedge d)$$

where a, b, c, and d are boolean, and $\oplus$ indicates exclusive-or. There are only 16 possible instances. An instance is an example of the concept if and only if $(a \vee b) \oplus (c \wedge d)$ is true for the given values of a, b, c, and d in the instance. This problem was chosen because the concept cannot be learned by a single LTU and because the subconcepts involve testing whether at least x of n variable are true, a kind of problem that is well suited to a LTU.

The standard perceptron learning algorithm repeatedly draws a training instance at random from the set of training instances, and presents it to the error correction procedure in use. A variant of the algorithm was employed here, in which the training instances were drawn in order from the entire space of 16, one after the other. The list of training instances is considered to be circular. This training strategy is not essential to the algorithm.

The following training procedure was employed: while the perceptron tree fails to classify all sixteen instances correctly, apply the perceptron tree error correction procedure to the next training instance. Figure 1 shows the percentage of the sixteen training instances classified correctly after training on the next training instance. The first split occurred while training on the 34th instance. Classification performance temporarily dipped to 0% when the perceptron tree consisted of a decision node with no branches. As the branches were grown on subsequent training, performance was generally better than before the split. The second split came while training on the 60th instance. Classification performance temporarily dipped to 44% because one of the leaf nodes was a decision node with no branches. As the branches below that node were grown during subsequent training, performance climbed to 100% after the 94th instance, at which point the concept had been learned perfectly.

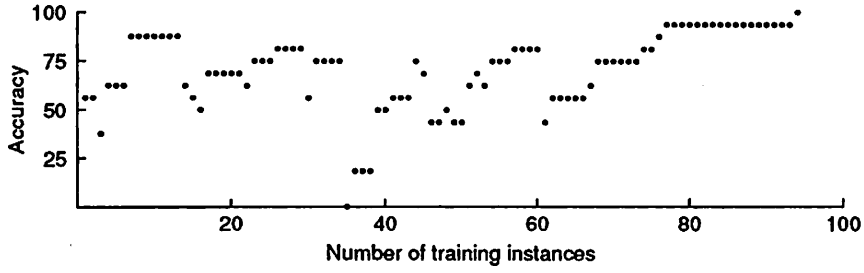

Figure 1. Learning curve for perceptron tree.

Figure 2 shows the final perceptron tree. It contains two decision nodes and three LTU nodes. For each LTU, the **W** and θ are depicted as a simple matrix in which the row is indexed by the value of a variable and the column is indexed by the name of the variable. To illustrate the LTU notation, how a symmetric LTU operates, and how a

perceptron tree is used to classify an instance, consider how the instance ($a=F$, $b=F$, $c=T$, $d=F$) is classified. Because c is the test attribute at the root, and $c=T$ in the instance, the T branch is taken. Because d is the test attribute at the subtree, and $d=F$ in the instance, the F branch is taken. Now, at the LTU node, the instance is encoded as 1 for each feature present and -1 for each feature absent. Thus $\mathbf{W}\cdot\mathbf{I}=-5$, because

W	*a*	*b*	*θ*
F	−2	−1	
T	1	2	1

·

I	*a*	*b*	*θ*
F	1	1	
T	−1	−1	1

$= -5,$

so the instance is classified as negative.

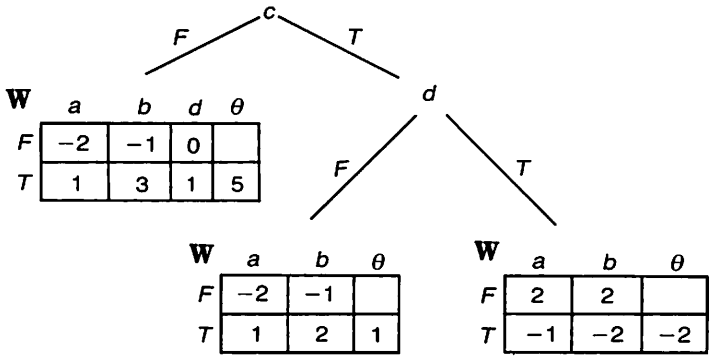

Figure 2. Perceptron tree.

Figure 3 shows the plain decision tree that would be built by Quinlan's ID3. Note that it has eight decision nodes and nine answer nodes.

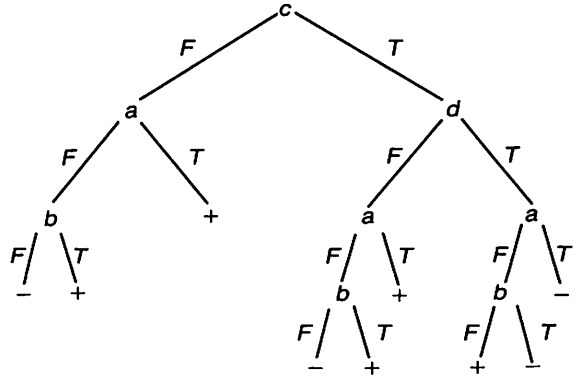

Figure 3. Decision tree.

5. Discussion

This section discusses several aspects of perceptron tree learning and how the algorithm is being improved.

5.1. Learning Behavior

Because a perceptron tree has a LTU at each leaf node, much of the learning behavior is characteristic of plain perceptron learning. Only when a LTU is discarded in favor of an attribute test is learning behavior different. Classification accuracy may drop suddenly when a new LTU is initialized. This is most noticeable at the root because the sole LTU is responsible for classifying all instances. As the tree develops, this effect is less noticeable because each LTU is responsible for classifying a smaller overall portion of the instances. When a LTU is replaced with an attribute test, the other LTUs are not affected.

5.2. Initializing a Linear Threshold Unit

One question that arises is how best to initialize the weights of a LTU. This is of no particular concern for a LTU at the root, but when a LTU is to be replaced by an attribute test, it seems a shame simply to discard the LTU. An alternative would be to save it and initialize each new child LTU below the attribute test with the corresponding weights of the old parent LTU. Each new child LTU would be based on the features of the old parent LTU, less those features that are based on the new attribute test at the parent node. Unfortunately, this alternative is unfounded. There is no reason to believe that the resulting projection is any better than some other initial set of weights. However, if one were to keep track of the best $\mathbf{W}$ ever seen, as per Gallant's (1986) Pocket Algorithm, then initializing a new LTU in terms of these weights would be founded. This idea is currently being tested.

5.3. Training and Sample Complexity

Training instances must be presented repeatedly because a LTU is not guaranteed to remain consistent with the previously observed training instances, unless it is correct for all the instances. Two questions arise. First, how many distinct instances constitute an adequate training set? Second, how many times must these instances be presented until the algorithm finds a consistent perceptron tree?

Duda & Hart (1973) point out that the capacity of a hyperplane is twice the number of attributes, and that one would like 'several times' this number of training instances in order to ensure that the hyperplane is not underfitting the data. Assume there are d attributes, each with b possible values, for a total of bd features. Thus, one needs to train on kdb, $k>2$, instances for any given LTU. At the root of the perceptron tree, this is simply kdb training instances. If the root is replaced by an attribute test, then there will be as many as b new LTUs, each based on $d-1$ attributes, thus requiring $b[k(d-1)b]$ training instances, assuming $k(d-)b$ instances for each of the b branches. It is apparent that as the tree develops, the required number of training instances to prevent underfitting can increase. In the worst case, one needs

$$\max\{b^0k(d-0)b, \ldots, b^ik(d-i)b, \ldots, b^{d-1}k[d-(d-1)]b\}$$

distinct training instances, which will typically be kb^d.

Regarding sample complexity, Hampson & Volper (1986) have shown empirically that the number of weight adjustments is exponential in the number of features bd. Assume that the base is some constant c. Then one would expect $O(c^{bd})$ weight adjustments at the root. This is probably too high, given the perceptron tree algorithm's aggressive heuristic for when to split. Nevertheless, if the root LTU is replaced

by an attribute test, then there will be b LTUs, each requiring $O(c^{b(d-1)})$ weight adjustments. In the worst case, the number of weight adjustments is

$$O\left(\sum_{i=0}^{d-1} b^i c^{b(d-i)}\right) = O\left(\frac{c^{b(d+1)} - c^b bd}{c^b - b}\right)$$
$$= O(c^{bd}).$$

Thus, the number of weight adjustments for perceptron tree learning is of the same order of complexity as plain perceptron learning.

5.4. Noise Handling

It has been assumed that the training instances are labeled consistently. This may be unnecessarily strong, as a LTU can find a solution vector in spite of a certain amount of mislabeling. One question is whether there is so much noise that the error correction procedure will be fooled into splitting the space prematurely. A second question is whether there is so much noise that the counts for the number of positive and number of negative instances for each of the w_i come to be unrepresentative of the true instance space, in which case the information-theoretic splitting heuristic will be fooled into picking a poor attribute test for the decision node. Noise resilience of the perceptron tree error correction procedure has not yet been examined.

5.5. Restriction to Discrete-valued attributes

The current algorithm is restricted to discrete-valued attributes. This is because every attribute has the potential to be selected as the test at a decision node. One needs a small number of branches at a decision node. A branch for each observed value of a numeric variable would be unwieldy and would produce poor generalization. A variety of techniques exist for partitioning a range of numeric values into a small set of subintervals. This allows mapping a numeric variable to a discrete variable, by determining into which interval a given value falls. One could extend the perceptron tree formalism to permit numeric variables. For example, within a LTU, use the numeric value directly; do not map it to presence or absence of a feature. Within a decision node, determine a mapping to intervals and use this dynamically created discrete variable. This approach has its own problems, principally that it still attempts to split the space based on the value of a single variable. The next section discusses an alternative approach that also addresses this problem.

5.6. Splitting Criteria

A significant shortcoming of the algorithm is that the split at a decision node is based on a single variable. Consider the concept defined by the half-plane $\{(x, y) | x < y\}$. Because the classification depends on a relationship between the two variables, no split on either one of them alone is informative. The half-plane would need to be approximated by a union of quarter-planes, each aligned with the x and y axes.

Breiman *et al.* (1984) make this point that considering only single variable splits may be inappropriate and can lead to overly large trees that do not represent the concept well. They call for splits on linear combinations of the (numeric) variables (Qing-Yun & Fu, 1983). One chooses variables for the combination, constructs a LTU in terms of these variables, and then places the LTU at the decision node. Thus, the

LTU is a binary variable. The result is a piecewise-linear classifier. Breiman *et al.* only allow linear combination splits over numeric variables, but there is no reason to believe that the technique would not work with binary features or a mix of numeric and binary features. The next version of the perceptron tree formalism and learning algorithm will incorporate linear combination splits. Numeric variables will be used directly, and not be recoded as presence or absence of particular values. By searching for a small set of variables on which such a linear combination is to be based, the sequential testing of attributes will be retained, which is one of the advantages of decision trees.

5.7. Multi-layered Networks

There has been a great deal of recent study of multi-layered networks, particularly three-layered feed-forward networks in which the output of each and every unit at layer i is connected as an input to each and every unit at layer $i+1$. A natural question is how connectionist learning in such nets would compare to learning a perceptron tree. Such a comparison is difficult, largely because learning in a layered network is not as robust. One must choose a network architecture by hand, and then hope for good learning performance. If there are too many units in the net, then it will underfit the training data, giving poor generalization. If there are too few units, the network will have insufficient capacity to represent the concept. Compounding the network architecture problem is the fact that there is no known algorithm for training a multi-layered network that is guaranteed to find weights that cause the net to separate the training instances. The perceptron tree algorithm is not subject to these problems as it grows its data structure according to need.

Three recent studies (Mooney *et al.*, 1989; Weiss & Kapouleas, 1989; Fisher & McKusick, 1989) compared back propagation to several other learning algorithms, including plain decision tree induction. No single algorithm was best for all problems. All three studies observed that back propagation was by far the slowest of all algorithms tested. The studies also mentioned the difficulties of picking network architecture, learning rate, and momentum terms.

A recent development in connectionist learning is Ash's (1989) Dynamic Node Creation algorithm for adding new units to the hidden layer of a three-layered network. Although this allows dynamic growth in the hidden layer, it does not speak to the convergence problem. Also, learning with dynamic node creation is slower than plain back propagation.

5.8 Other Hybrid Approaches

Schlimmer (Schlimmer, 1987) has constructed a hybrid representation and associated learning algorithm embodied in his STAGGER program. The program maintains a pair of weights for each boolean term in his concept description. One corresponds to logical sufficiency, the other to logical necessity. By adjusting the weights and by adding or removing Boolean terms, the program searches for a consistent concept description. A recent addition to STAGGER is the ability to group values of a real-valued attribute into a dynamically formed interval, which constitutes a new boolean term that can become part of the concept description.

6. Conclusions

The perceptron tree representation and the perceptron tree error correction procedure can be seen either as a method for perceptron learning even when the space of instances is not linearly separable, or as a method for incremental construction of a

tree structure that is very much like a decision tree. The algorithm is incremental, does not save training instances, and is guaranteed to find a consistent concept description for all problems in which the instances are labeled consistently.

The work has been motivated by the specific need for an efficient incremental learning algorithm, and by the observation that the inherent biases in the formalisms of two particular learning algorithms are highly complementary. The ease of incrementally training a linear threshold unit complements the difficulty of incrementally building a decision tree. The ability to represent any concept in the decision tree formalism complements the inability to represent non-linearly separable concepts in the hyperplane formalism. The combination of complementary formalisms into a hybrid makes it possible to draw on the particular strengths of each of the individual formalisms. The case study reported here demonstrates that a perceptron tree representation retains the advantages of both the decision tree representation and the hyperplane representation, while shedding the major disadvantages. The approach has illustrated that it can be advantageous to devise a control strategy for drawing on a variety of complementary techniques instead of just one.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. IRI-8619107, a General Electric Faculty Fellowship, and by the Office of Naval Research through a University Research Initiative Program, under contract number N00014-86-K-0764. Helpful comments on earlier versions of this article were provided by Andy Barto, Pat Langley, David Haussler, Carla Brodley, Sharad Saxena, Peter Heitman, Margie Connell, Jamie Callan, Kishore Swaminathan, Victor Coleman, and Richard Yee.

References

Ash, T. (1989) *Dynamic node creation in backpropagation networks,* ICS Report 8901. San Diego, CA: University of California, Institute of Cognitive Science.

Breiman, L., Friedman, J.H., Olshen, R.A. & Stone, C.J. (1984) *Classification and Regression Trees.* Belmont, CA: Wadsworth International Group.

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

Fisher, D.H. & McKusick, K.B. (1989). An empirical comparison of ID3 and back-propagation. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence,* pp. 788–793. Detroit, MI: Morgan Kaufmann.

Fu, K.S. (1968) *Sequential Methods in Pattern Recognition and Machine Learning.* New York: Academic Press.

Gallant, S.I. (1986) Optimal linear discriminants. *Proceedings of the International Conference on Pattern Recognition,* pp. 849–852. IEEE Computer Society Press.

Hampson, S.E. & Volper, D.J. (1986) Linear function neurons: structure and training. *Biological Cybernetics,* 53, 203–217.

Hinton, G.E. (1989) Connectionist learning procedures. *Artificial Intelligence,* 40, 185–234.

Ho, Y.C. & Kashyap, R.L. (1965) An algorithm for linear inequalities and its applications. *IEEE Transactions on Electronic Computers,* EC-14, 683–688.

Lewis, P.M. (1962) The characteristic selection problem in recognition systems. *IRE Transactions on Information Theory,* IT-8, 171–178.

Michalski, R.S. & Chilausky, R.L. (1980) Learning by being told and learning from examples: an experimental comparison of the two methods of knowledge acquisition in the context of developing an expert system for soybean disease diagnosis. *Policy Analysis and Information Systems,* 4, 125–160.

Mingers, J. (1989) An empirical comparison of selection measures for decision-tree induction. *Machine Learning,* 3, 319–342.

Minsky, M. & Papert, S. (1972) *Perceptrons: An Introduction to Computational Geometry* (expanded edition). Cambridge, MA: MIT Press.

Mitchell, T.M. (1978) *Version spaces: an approach to concept learning.* Doctoral dissertation. Palo Alto, CA: Department of Electrical Engineering, Stanford University.

Mooney, R., Shavlik, J., Towell, G. & Gove, A. (1989) An experimental comparison of symbolic and connectionist learning algorithms. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence,* pp. 775–780. Detroit, MI: Morgan Kaufmann.

Moret, B.M.E. (1982) Decision trees and diagrams. *Computing Surveys,* **14,** 593–623.

Nilsson, N.J. (1965) *Learning Machines.* New York: McGraw-Hill.

Qing-Yun, S. & Fu, K.S. (1983) A method for the design of binary tree classifiers. *Pattern Recognition,* **16,** 593–603.

Quinlan, J.R. (1983) Learning efficient classification procedures and their application to chess end games. In R. S. Michalski, J. G. Carbonnel & T. M. Mitchell (Eds) *Machine Learning: An Artificial Intelligence Approach.* San Mateo, CA: Morgan Kaufmann.

Quinlan, J.R. (1986) Induction of decision trees. *Machine Learning,* **1,** 81–106.

Rumelhart, D.E. & McClelland, J.L. (1986) *Parallel Distributed Processing.* Cambridge, MA: MIT Press.

Schlimmer, J.C. & Fisher, D. (1986) A case study of incremental concept induction. *Proceedings of the Fifth National Conference on Artificial Intelligence,* pp. 496–501. Philadelpha, PA: Morgan Kaufmann.

Schlimmer, J.C. (1987) Incremental adjustment of representations. *Proceedings of the Fourth International Workshop on Machine Learning,* pp. 79–90. Irvine, CA: Morgan Kaufmann.

Shannon, C.E. (1948) A mathematical theory of communication. *Bell System Technical Journal,* **27,** 379–423.

Utgoff, P.E. (1986) Shift of bias for inductive concept learning. In R. S. Michalski, J. G. Carbonnell & T. M. Mitchell (Eds) *Machine Learning: An Artificial Intelligence Approach.* San Mateo, CA: Morgan Kaufmann.

Utgoff, P.E. (1989) Incremental induction of decision trees. *Machine Learning,* **4,** 161–186.

Utgoff, P.E. & Heitman, P.S. (1988) Learning and generalizing move selection preferences. *Proceedings of the AAAI Symposium on Computer Game Playing,* pp. 36–40. Palo Alto, CA.

Weiss, S.M. & Kapouleas, I (1989). An empirical comparison of pattern recognition, neural nets, and machine learning classification methods. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence,* pp. 781–787. Detroit, MI: Morgan Kaufman.