Machine Learning, 16, 161–183 (1994)

Learning Nonoverlapping Perceptron Networks from Examples and Membership Queries

THOMAS R. HANCOCK hancock@learning.siemens.com
Siemens Corporate Research, 755 College Road East, Princeton, NJ 08540

MOSTEFA GOLEA golea@physics.uottawa.ca
Ottawa-Carleton Institute for Physics, University of Ottawa, Ottawa, Ont., Canada K1N 6N5

MARIO MARCHAND mario@physics.uottawa.ca
Ottawa-Carleton Institute for Physics, University of Ottawa, Ottawa, Ont., Canada K1N 6N5

Editor: Leonard Pitt

Abstract. We investigate, within the PAC learning model, the problem of learning nonoverlapping perceptron networks (also known as read-once formulas over a weighted threshold basis). These are loop-free neural nets in which each node has only one outgoing weight. We give a polynomial time algorithm that PAC learns any nonoverlapping perceptron network using examples and membership queries. The algorithm is able to identify both the architecture and the weight values necessary to represent the function to be learned. Our results shed some light on the effect of the overlap on the complexity of learning in neural networks.

Keywords: Neural networks, PAC learning, nonoverlapping, read-once formula, learning with queries

1. Introduction

Despite the excitement generated recently by neural networks, learning in these systems has proven to be very difficult from a theoretical perspective (Blum and Rivest, 1988; Judd, 1988; Kearns and Valiant, 1988; Lin and Vitter, 1991). For this reason researchers have looked for positive results by considering restricted classes of neural networks (Baum, 1990a; Lin and Vitter, 1991), by providing the learning algorithm with additional information in the form of queries (Baum, 1991), or by restricting the distribution of examples (Baum, 1990b).

In this paper, we investigate the problem of learning the class of "*nonoverlapping*" perceptron networks (this terminology comes from Barkai, Hansel, and Kanter (1990) and Barkai and Kanter (1991)). These are loop-free neural nets in which each node, including the input units, has only one outgoing non-zero weight (figure 1(a)). This class of representations includes as a subclass nonoverlapping multilayer networks (figure 1(b)) and nonoverlapping cascade networks. Such networks, in which each node has fan-out 1, are also referred to in the literature as *read-once formulas*. Our work is partly motivated by, and uses techniques from, recent positive results for learning other classes of read-once formulas (Angluin, Hellerstein, and Karpinski, 1993; Bshouty, Hancock, and Hellerstein, 1992a; Bshouty, Hancock, and Hellerstein, 1992b; Goldman, Kearns, and Schapire, 1990; Kearns, Li, Pitt, and Valiant, 1987; Pagallo and Haussler, 1989; Schapire, 1991). In the terminology of that literature, nonoverlapping perceptron networks are read-once formulas (or synonymously μ formulas) over the basis of weighted threshold functions.

One can think of this type of architecture as a network of "*decoupled*" perceptrons, which in terms of architecture complexity lies somewhere between the single perceptron and the traditional feed forward neural net. As such, studying this restricted class may shed some light on the gap that exists, in terms of computational complexity, between training single perceptrons, which can be done in polynomial time (Karmarkar, 1984) and training feed forward nets, which has been proven to be intrinsically hard (Blum and Rivest, 1988; Judd, 1988), even if the algorithm is allowed to represent its hypothesis in ways other than as a feed forward network (Kearns and Valiant, 1989; Angluin and Kharitonov, 1991; Kharitonov, 1993).[1] Of fundamental importance is the question of whether or not removing the overlap between the receptive fields of the nodes makes the learning problem easier.

Standard techniques (Kearns, Li, Pitt, and Valiant, 1987) show that the problem of learning nonoverlapping networks from only examples drawn according to an arbitrary distribution is no easier than the problem where the input variables may have an arbitrary number of outgoing weights. Kearns and Valiant have shown that this (seemingly) more general problem is intractable (Kearns and Valiant, 1989). Thus to achieve interesting results we must consider a slightly easier learning model. We allow the algorithm to make membership queries, in which the learner supplies an *instance* a to an oracle (perhaps a human expert) and is told its classification $f(a)$. This seems a reasonable extension since people tend to use queries in learning. With membership queries the problem of learning general perceptron networks remains intractable (Angluin and Kharitonov, 1991), but the nonoverlapping case becomes easier.

There are a number of previous algorithms for learning other classes of read-once formulas once membership queries are allowed (Angluin *et al.*, 1993; Bshouty *et al.*, 1992a, 1992b; Goldman *et al.*, 1990). Angluin, Hellerstein, and Karpinski (1993) give an algorithm that learns boolean read-once formulas over gates computing AND, OR, and NOT. Bshouty, Hancock, and Hellerstein (1992b) have generalized this to allow gates that compute any function that has a constant number of inputs or that is symmetric (including those perceptrons that assign weights of equal magnitude to every input). Bshouty *et al.* (1992a) also give a membership query algorithm that learns non-boolean arithmetic read-once formulas, introducing the perceptron like ability to compute weighted sums (but not to take thresholds).

Motivated by this relevant work, we study the learnability of read-once perceptron networks from examples and membership queries. We adopt Valiant's PAC model (Valiant, 1984) as our criterion for learning. We use an Occam algorithm (Blumer, Ehrenfeucht, Haussler, and Warmuth, 1989) that achieves learning by drawing a certain number of random examples and then fitting this sample with a consistent nonoverlapping perceptron network. Our algorithm has the feature that all its membership queries are made on instances where each input variable is set to some value from its observed domain in the random sample. Thus the algorithm will work for any mix of real-valued, integer, boolean, etc. variables without requiring any prior knowledge of the variable type. Note that in contrast to most neural network learning algorithms, we do not assume the architecture of the network is known in advance. Rather, it is the task of the algorithm to find both the architecture of the net and the weight values necessary to represent the function to be learned. The class of nonoverlapping perceptron networks either generalizes or is incom-

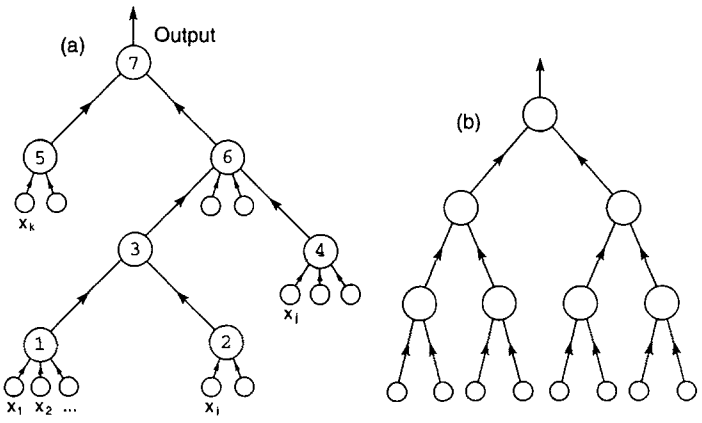

Figure 1. (a) Architecture of a nonoverlapping perceptron network. Note that each node, including the input variables but excluding the output, has only one outgoing connection. As examples of the definitions given in Section 2, nodes 1 through 6 are hidden units, node 7 is the root and output unit, nodes 1, 2, 4, and 5 are bottom level units, the parent of x_1 and x_2 is node 1, variables x_1 and x_2 are siblings, variables x_1 and x_i are descendants of node 6, and the children of node 6 are node 3 and node 4. (b) A nonoverlapping layered network.

parable to all the known learnable classes of read-once formulas. Unlike perceptrons, none of those classes allow nodes that compute functions that are both asymmetric and have an unbounded number of inputs.

The main contribution of this paper is a polynomial time algorithm that PAC learns any nonoverlapping perceptron network from examples and membership queries under an arbitrary distribution of examples. The paper is organized as follows: In Section 2 we introduce some notation and terminology. In Section 3 we review the PAC learning model and calculate an upper bound on the number of examples needed to learn nonoverlapping perceptron networks. In Section 4 we show how without loss of generality we may consider simplified versions of the problem. Section 5 handles the problem of learning when the architecture is known in advance, and it is intended primarily as an introduction to the general case. In Section 6 we address the main problem of learning both the architecture and weights necessary to represent our target function. We analyze our algorithm in Section 7. In Section 8 we summarize our results and discuss future research directions.

2. Definitions

In this section we introduce some notation and terminology that we use throughout this paper. Examples for some of the terms defined here are given in figure 1(a).

A *linear threshold function* on a set X of n variables is specified by a vector of n real valued weights (w_i) and a single real valued threshold (θ). The output of the function is 1 if the following inequality holds, and 0 otherwise:

$$\sum_{x_i \in X} w_i x_i \geq \theta.$$

Such functions are also referred to as *perceptrons* or *half-spaces*.

A Nonoverlapping Perceptron Network (hereafter NPN) over a set of input variables X can be viewed as a rooted tree (figure 1(a)). The root is the *output unit*. Each internal node, or *computation unit*, of the tree (including the root) is labeled with a perceptron that has one input variable corresponding to each of the node's children in the tree. The computation units other than the root are called *hidden units*. Each leaf in the tree is labeled with a variable from X in such a manner that no variable appears on more than one leaf. The network is evaluated by substituting values for the input variables and then propagating these values to the root (output unit) in the usual manner. A NPN can also be referred to as a read-once formula over the basis of linear threshold functions (or perceptrons). We refer to a hidden unit whose children are all leaves as a *bottom level unit*.

The *parent* of a node (or a variable) is the computation unit to which the node's output is an immediate input. Every node is a *child* of its parent. We say two variables are *siblings* if they share a common parent. We say a node (or variable) is a *descendant* of a computation unit if the node appears in the subtree rooted at that unit.

Let $X = \{x_1, x_2, \ldots, x_n\}$ be the set of the n input variables. An *assignment* to X on a domain D is a mapping from X to D, where D may be $\{0, 1\}$, the real numbers $\Re$, or some other subset of $\Re$. We denote such an assignment a by $(a_1, a_2, \ldots, a_n)$, where each $a_i \in D$ is the value assigned to x_i $(i = 1, 2, \ldots, n)$. For a NPN f defined on X, we let $f(a)$ denote the output of the network when each input variable is set to its value in assignment a. An *example* is an ordered pair $\langle a, f(a) \rangle$. If $f(a) = 1$, we say a is a *positive* example. If $f(a) = 0$, we say a is a *negative* example. A *sample* is a set of examples.

A *partial assignment* to X is a mapping from X to $D \cup \{*\}$, where the value "$*$" signifies being unassigned. For example, $(*, \ldots, *, p_i, \ldots, p_n)$ denotes a partial assignment that assigns values to $x_i, \ldots, x_n$, leaving $x_1, \ldots, x_{i-1}$ unassigned. Let p be a partial assignment to X. We denote by p/a the assignment to X obtained by setting p's unassigned variables according to a. So if $p = (*, \ldots, *, p_i, \ldots, p_n)$ and $a = (a_1, a_2, \ldots, a_n)$, then $p/a = (a_1, \ldots, a_{i-1}, p_i, \ldots, p_n)$.

If $x^{(0)}$ is a value from the domain of a variable x_i we shall write $a_{x_i \leftarrow x^{(0)}}$ to denote the assignment that is identical to a on all variables except x_i, on which it evaluates to $x^{(0)}$. If W is a set of variables we shall use $a_{W \leftarrow x^{(0)}}$ to denote the assignment obtained from a by setting all variables in W to $x^{(0)}$, and we shall use $a_{W \leftarrow b}$ (for some other assignment b) to denote the assignment obtained from a by setting each variable $x_i \in W$ to b_i.

If f is a function defined on X, each partial assignment p on X induces a *projection*, f_p, which is the function obtained from f by replacing by the appropriate constants those variables in f to which p assigns values (so $f_p(a) = f(p/a)$). Note that the class of NPNs is *projection closed*, meaning that any projection of a NPN can also be represented as a NPN (by absorbing the constants placed at leaves into their parents' threshold values).

3. The learning model and sample complexity

Intuitively, an efficient learning algorithm is one that, given a "reasonable" number of examples labeled according to an unknown function, is "likely" to produce a "good" approximation of the unknown function after a "reasonable" amount of time. The unknown (target) function may be any from some known class of functions. We adopt Valiant's formalization of this intuitive notion into what is known as the Probably Approximate Correct (PAC), or Distribution Free, model of learning (Valiant, 1984; Blumer *et al.*, 1989).

Definition 1. Let F be a class of boolean functions defined over an n-dimensional input space. F is said to be PAC learnable from examples if there exists an algorithm A such that for any target function $f \in F$, for any $0 < \epsilon < 1$, and for any $0 < \delta < 1$ the following holds:

Given n, ϵ, δ as inputs, and access to examples generated according to a fixed but unknown probability distribution P over the instance space, the algorithm runs in time polynomial in $(n, 1/\epsilon, 1/\delta)$ and produces a hypothesis h from F that with a probability at least $1 - \delta$ disagrees with f on a future example generated according to P with probability at most ϵ.

If the algorithm uses also membership queries, F is said to be PAC learnable from examples and membership queries.

The *sample complexity* of the learning algorithm is the number of random examples it draws.

A central concept of the PAC learning model is the *Vapnik Chervonenkis (VC) dimension* of a class of functions. Intuitively, the VC dimension is a measure of how powerful a class of functions is in terms of the size of the largest sample for which any split between positive and negative examples is realized by some function in the class. The following notation is from Baum and Haussler (1989).

Definition 2. Let F be a set of functions mapping $\Re^n$ to $\{0, 1\}$. For a set $S \in \Re^n$, let $\Delta_F(S)$ denote the number of distinct dichotomies of S induced by functions $f \in F$. Let $\Delta_F(m)$ denote the maximum of $\Delta_F(S)$ over all $S \in \Re^n$ of cardinality m. The *Vapnik Chervonenkis (VC) dimension* of F, denoted by $VCdim(F)$, is the value of the largest m for which $\Delta_F(m) = 2^m$ (i.e. the largest m for which F induces all possible dichotomies on some set of size m).

The following lemma shows that the PAC learning problem can be reduced to finding a hypothesis consistent with a polynomial number of examples. (All logarithms in this paper are taken base 2, and e is the base for the natural logarithm.)

LEMMA 1 (Blumer *et al.*, 1989). *Let F be a non-trivial, well-behaved*[2] *class of functions mapping $\Re^n$ to $\{0, 1\}$. Then for any $0 < \epsilon, \delta < 1$ and any sample of at least*

$$m_0 = \max\left(\frac{4}{\epsilon}\log\frac{2}{\delta}, \frac{8\,VCdim(F)}{\epsilon}\log\frac{13}{\epsilon}\right)$$

randomly selected examples, the probability that any function from F that is consistent with those examples has error at most ϵ is at least $1 - \delta$.

In the following section we calculate an upper bound for the VC dimension of the class of NPNs. From this we derive a sample complexity sufficient to achieve the PAC learning criterion.

3.1. *The VC dimension of NPNs*

In order to derive the sample size necessary to achieve PAC learning, we appeal to a result from the literature. Lemma 2 is adapted from Corollary 3 of Baum and Haussler (1989).

LEMMA 2 [Baum and Haussler, 1989]. *Let F be the class of all functions computed by feed forward nets defined on a fixed underlying graph G with E edges and $N \geq 2$ computation nodes, each of which computes a linear threshold function. Let $W = E + N$ (the total number of weights in the network, including one weight per edge and one threshold per computation node). Then $\Delta_F(m) \leq (Nem/W)^W$ for all $m \geq W$ and $VCdim(F) \leq 2W \log(eN)$.*

A NPN has at most $2n - 1$ computation nodes (without loss of generality). That follows since at most $n-1$ of the nodes can have more than one input, and we may assume that single input units appear only at the bottom level above one of the n input variables (single input units are trivial if their input is boolean). Each node has an outgoing weight and a threshold, so there are at most $4n - 2$ weights and thresholds (we may assume the input weight for any single input perceptron is 1). We cannot, however, immediately apply Lemma 2 with $N = 2n-1$ and $W = 4n-2$, because there is no fixed underlying graph known in advance. Instead we use the result to prove the following lemma, bounding the VC dimension for our class of functions.

LEMMA 3. *The class of nonoverlapping perceptron networks with n inputs has VC dimension at most $13n \log(2en) + 4n \log\log(4n)$.*

Proof: Let $\mathcal{G}$ be a set of graphs each with E edges and $N \geq 2$ computation nodes. Let $F^{(\mathcal{G})}$ be the class of all functions computed by feed forward nets defined on *any* underlying graph $G \in \mathcal{G}$. Since no single graph in $\mathcal{G}$ can induce more than $\Delta_F(m)$ dichotomies on a sample of size m, it follows that $\Delta_{F^{(\mathcal{G})}}(m) \leq |\mathcal{G}|\Delta_F(m)$. For $N > 2$ and $W = E + N$, it is easily verified that for

$$m = \log(|\mathcal{G}|) + W \log\log(|\mathcal{G}|) + 2W \log(Ne) \tag{1}$$

we have $2^m > |\mathcal{G}|(Nem/W)^W$. Therefore by Lemma 2, $2^m > |\mathcal{G}|\Delta_F(m)$ and hence, as argued above, $2^m > \Delta_{F^{(\mathcal{G})}}(m)$. This states that $F^{(\mathcal{G})}$ cannot shatter a set of m points, giving us the result that $VCdim(F^{(\mathcal{G})}) \leq m$.

Any NPN on n variables can be expressed on an underlying graph generated by 1) picking a binary tree over n leaves (and adding unary gates at the leaves), 2) assigning each of the n variables to a leaf, and 3) deciding for each of the $n-2$ non-root internal nodes whether to

merge it with its parent, creating a single unit of larger fan-in. Thus we bound the number of possible underlying graphs for a NPN as follows (the first two terms are the Catalan number, counting binary trees with n leaves):

$$\begin{aligned} |\mathcal{G}| &\leq \frac{1}{n}\binom{2n-2}{n-1}\ n!\ \ 2^{n-2} \\ &\leq \frac{(2n-2)!}{(n-1)!}2^{n-2} \\ &\leq (4n)^{n-1}. \end{aligned}$$

Substituting this bound on $|\mathcal{G}|$ along with $N \leq 2n-1$ and $E = 2n-1$ into equation (1), we get

$$\begin{aligned} VCdim(NPN) &\leq (n-1)\log(4n) + W\log((n-1)\log(4n)) + 2W\log(Ne) \\ &\leq (n-1)\log(4n) + (4n-2)\log((n-1)\log(4n)) \\ &\quad + (8n-4)\log(e(2n-1)) \\ &\leq 13n\log(2en) + 4n\log\log(4n). \end{aligned}$$

■

Substituting this VC dimension bound into Lemma 1 gives us the sample complexity result we need ($m = \tilde{O}(\frac{n}{\epsilon})$).

COROLLARY 1. *If a nonoverlapping perceptron network h is correct on a sample of at least*

$$m = \max\left(\frac{4}{\epsilon}\log\frac{2}{\delta}, \frac{104n\log(2en) + 32n\log\log(4n)}{\epsilon}\log\frac{13}{\epsilon}\right)$$

randomly selected examples, then with probability at least $1-\delta$ the hypothesis h has error less than ϵ.

This result solves the statistical aspect of our learning problem. We can now concentrate on the computational problem of using membership queries to fit a NPN to a sample of m examples.

4. Some simplifying reductions

In this section we describe some standard reductions that allow us to make simplifying assumptions about the form of our target function and our sample.

Suppose f is a NPN over the variable set X, and M is a set of examples classified according to f. For each variable $x_i \in X$ we define the *induced domain* on M to be the set of values that x_i assumes in M. While the true domain of x_i may be continuous and/or unbounded, our algorithm need only consider input settings from the induced domain. This is a discrete set of values, so we may speak of the minimum and maximum value of a variable. We shall say a sample is *normalized* if each variable assumes a minimum value of 0 and a maximum value of 1. We may assume without loss of generality that this is the case. We can normalize our sample by applying a linear scaling function for each variable,

mapping its induced domain to $[0, 1]$, and then inverting this scaling function whenever we make a membership query. But it is perhaps cleaner just to consider "0" and "1" settings for variables in the remainder of the paper to be notation for the smallest and largest values that the variable assumes in the sample. In our arguments we make repeated reference to the most *influential* input to a hidden unit. This is the input for which the product of its weight times the difference between its "1" and "0" values is largest (i.e. the swing in the weighted sum obtained by flipping this input between its 0 and 1 values is maximal).

A *justifying assignment* for a variable is an assignment for which changing the value of that variable changes the value of the target function (i.e. $f(a_{x_i \leftarrow 1}) \neq f(a_{x_i \leftarrow 0})$). We can apply a standard procedure to our sample to obtain a justifying assignment for each variable (Angluin *et al.*, 1993; Hancock, 1991). This procedure checks each variable x_i to observe whether we can fix x_i to 0 in every sample point without changing any of those examples' classifications (determined by making membership queries). If there is such an "irrelevant" x_i, we instead learn the projection f_p of f that forces x_i to 0 (in effect removing x_i from the set of variables). Since the class of NPNs is projection closed, this modified target f_p is still realizable as a NPN. All future membership queries are made not on f, but on f_p (i.e. we intercept a query instance before passing it to the oracle and set x_i to 0). The learning goal is now to find a NPN consistent with f_p on our sample (from which we can now discard variable x_i). Any such NPN not containing the eliminated irrelevant variable will be consistent with the true function. We continue eliminating irrelevant variables until we can do so no longer. At this point either none are left (in which case a constant function is a consistent hypothesis), or else some subset of variables remain. The fact that those variables cannot be eliminated means that the sample contains a justifying assignment for each.

A useful property of NPNs is that we may assume without loss of generality that the perceptrons within our target network f contain negative weights only for those inputs that lead directly from variables (rather than deeper perceptrons). This is by an analog to De Morgan's laws that allows us to push negation of weights down to the leaves. Consider a perceptron in f that has some term $-w_i x_i$ in its weighted sum (where $w_i > 0$). We can rewrite this term as $w_i(1 - x_i) - w_i$. Thus we can eliminate the negative weight by adding w_i to the threshold and replacing the x_i input by its logical negation. We complement x_i's input by taking the perceptron whose output is x_i and multiplying all its weights and thresholds by -1 (this also requires changing the comparison from $\geq$ to $>$, which we can allow, or which we can simulate by adding a sufficiently small ϵ to the threshold). Note that while this process may introduce new negative weights, it does so at a lower level. By repeating this process all negative weights will be pushed to the input level.

We say a function f over X is *monotone* in a variable $x_i \in X$, if for all assignments a and any pair of values $x^{(1)} \geq x^{(0)}$, it is true that $f(a_{x_i \leftarrow x^{(1)}}) \geq f(a_{x_i \leftarrow x^{(0)}})$ (i.e. as x_i increases, f does not decrease). We say a function f over X is *anti-monotone* in a variable $x_i \in X$, if for all assignments a and any pair of values $x^{(1)} \geq x^{(0)}$, it is true that $f(a_{x_i \leftarrow x^{(1)}}) \leq f(a_{x_i \leftarrow x^{(0)}})$ (i.e. as x_i increases, f does not increase). We say a function is monotone if it is monotone in every variable. Since we assume f contains negative weights only on input variables, every variable has either no negative weights on its path to the root, in which case it is monotone, or it has exactly one, in which case it is anti-monotone. Since we have a justifying assignment for each variable, we can easily determine which is

the case. We may reduce our problem to the monotone case (i.e. no negative weights) by replacing each anti-monotone x_i with a new variable representing $(1 - x_i)$. Note that if a is a justifying assignment for x_i in M, and x_i is monotone, it follows that $f(a_{x_i \leftarrow 1}) = 1$ and $f(a_{x_i \leftarrow 0}) = 0$.

To summarize this section, we have argued that we may assume without loss of generality that the target NPN is monotone and contains no negative weights. Furthermore we may assume that our sample M is normalized and contains a justifying assignment for every variable. In what follows, whenever we refer to the target NPN and the sample M we suppose they satisfy these assumptions. Moreover, we assume that the target NPN has at least one hidden unit. The case where the target NPN has only one computation node (the output) is trivial.

5. Learning nonoverlapping perceptron networks: Known architecture

Let us assume for a moment that the architecture is fixed. In this case, the problem of learning reduces to that of *loading* a given set of examples in a given architecture, i.e. finding the weight values such that the given net is consistent with all the examples. We consider just a simple NPN with two nodes (figure 2).

The problem, often called the credit assignment problem (CAP), is to determine the output of the sub-function of each hidden unit on every example in the sample. Baum (1990a) suggested that no approach that avoids the CAP will work. Here we can solve this problem exactly by exploiting the fact that, because the receptive fields of the nodes are disjoint, some examples can be separated by *one and only one* node.

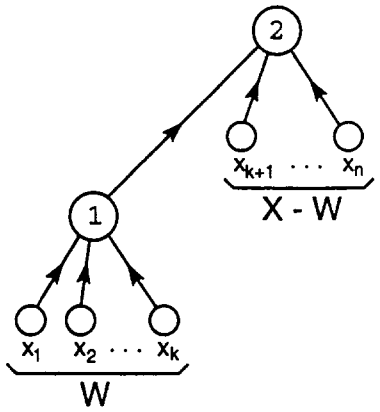

Figure 2. The nonoverlapping perceptron net considered in section 5. It consists of two nodes. Node 1 is connected to the set of variables W. Node 2 is connected to the set of variables $X - W$. Node 2 receives input also from node 1.

To see this, let $W = \{x_1, \ldots, x_k\}$ be the set of variables connected to node 1 and $(X - W) = \{x_{k+1}, \ldots, x_n\}$ the set of variables connected to node 2 (figure 2). Let $a = (a_1, a_2, \ldots, a_n)$ be a justifying assignment for a variable $x_i \in W$. By definition,

$$f(a_{x_i \leftarrow 1}) \neq f(a_{x_i \leftarrow 0}).$$

Define the partial assignment $p = a_{W \leftarrow *}$. Let b be an arbitrary input assignment for which we wish to calculate the output of node 1. We use a membership query to determine $f(p/b) = f(b_1, \ldots, b_k, a_{k+1}, \ldots, a_n)$. Note that the examples $a_{x_i \leftarrow 1}$, $a_{x_i \leftarrow 0}$, and p/b differ only on variables from W. The only node that can separate p/b from either $a_{x_i \leftarrow 1}$ or $a_{x_i \leftarrow 0}$ is node 1. The output of node 1 on example b will be the same as its output on p/b (b and p/b agree on all variables in W). This generalizes to give us the following fact for a nonoverlapping (and monotone) network.

LEMMA 4. *If f is a NPN over X, and p is a partial assignment that assigns values to exactly those variables that are not descendants of some unit G in f, then the projection f_p will either be a constant function or will be equivalent to the subnetwork of f rooted at G.*

Proof: Let g be the subnetwork rooted at G. Let f' be the NPN obtained from f by deleting G and its descendants and replacing them with a new variable (representing the output of g). Since f is nonoverlapping, no variable is an input to both g and f'. The projection f_p is computed by evaluating f' when all its inputs except g are fixed as in p. This projection of f' can either be a constant function (in which case so is f_p), or it can depend on the remaining input, g. Since f (and all its projections) are monotone, the only way f_p can be non-constant is if $f_p \equiv g$ (were f non-monotone, we might have $f_p \equiv \neg g$). ■

In other words, for any example $b = (b_1, b_2, \ldots, b_n)$ in our sample we can compute the induced output value from node 1 as $f_p(b) = f(b_1, \ldots, b_k, a_{k+1}, \ldots, a_n)$. Thus to learn the perceptron associated with node 1, we learn a function consistent with f_p (where $f_p(b) = f(p/b)$). Once this is done, we can learn the perceptron associated with node 2 over the set of variables $(X - W) \cup \{y\}$, where the new variable y represents the output of node 1. The method can be extended easily to an arbitrary NPN. This technique is taken from work in read-once formula learning (Bshouty *et al.*, 1992a and 1992b), where the architecture is termed the "skeleton" of the formula.

Before we leave this section, we prove the following consequence of our discussion. This will later prove useful as a criterion to rule out invalid architectures. We observe that changing an assignment b by modifying the variables (W) from some subnetwork does not affect $f(b)$, as long as the modified assignment induces the same output ($f_p(b)$) on that subnetwork. One such way we can change b without affecting $f_p(b)$ is to set the variables $W - \{x_i\}$ to agree with a justifying assignment a for x_i, and to set x_i to $f_p(b)$.

LEMMA 5. *If $W \subset X$ are those variables that appear in the subtree of some hidden unit of a NPN f, and if a is a justifying assignment for some $x_i \in W$, then for $p = a_{W \leftarrow *}$ it will be true for every example b that*

$$f(b) = f(b_{W-\{x_i\}\leftarrow a, x_i \leftarrow f_p(b)}).$$

Proof: Let $b' = b_{W-\{x_i\}\leftarrow a, x_i \leftarrow f_p(b)}$. Since b and b' agree on all variables in $X - W$ (those not appearing in W's sub-network), we can have $f(b) \neq f(b')$ only when $f_p(b) \neq f_p(b')$ (Lemma 4). But $f_p(b') = f(p/b') = f(a_{x_i \leftarrow f_p(b)})$. Since a is a justifying assignment for x_i (and f is monotone) this must equal $f_p(b)$. ■

6. Learning nonoverlapping perceptron networks: Unknown architecture

6.1. *Partitioning and the main routine*

The problem is to find a NPN consistent with a sample M. Our basic approach is to search for a *partition* of the variable set X into $W \cup (X - W)$ for which we can apply the fixed architecture solution of the previous section. We would like to find a set W that is exactly those variables that are descendants of some hidden unit of f (figure 3). A partition allows us to decompose our problem of finding a consistent hypothesis with f into the smaller problems of finding hypotheses consistent with NPNs g_1 and g_2. Both g_1 and g_2 can be expressed as projections of f. If a is a justifying assignment for $x_i \in W$, then $g_2 \equiv f_p$ for $p = a_{W \leftarrow *}$, and $g_1 \equiv f_q$ for $q = a_{(X-W)\cup\{x_i\}\leftarrow *}$ (where the x_i input represents g_1's input from the subnetwork over W).

We represent a *decomposition* of f as a three-tuple (x_i, a, W). Lemma 5 states that if W indeed contains exactly the variables that are descendants of some hidden unit, then every example b in our sample M will satisfy

$$f(b) = f(b_{W-\{x_i\}\leftarrow a, x_i \leftarrow f_p(b)}). \tag{2}$$

Note that by the definition of q this is equivalent to the condition

$$f(b) = f_q(b_{x_i \leftarrow f_p(b)}).$$

We say a decomposition is *valid* for M if it satisfies condition (2). We say a decomposition is *non-trivial* if $W \subset X$ and either $|W| > 1$ or else x_i (the only element of W) takes on more than two values in M. Our learning approach is divide and conquer, where we reduce the problem of finding a NPN consistent with f to that of finding NPNs consistent with the "simpler" targets f_p and f_q, determined as above according to a valid non-trivial decomposition. When we can break down the problem no further, we argue that a single perceptron can fit the sample. The idea that learning read-once formulas can be reduced to finding partitions according to non-trivial subformulas, and then learning single node formulas as a base case, is a common theme in read-once formula algorithms (Bshouty *et al.*, 1992a and 1992b). (Typically, as is the case here, the main work of the algorithm is to find the partitions.)

We have motivated the definition of a valid decomposition by considering the case where W is the set of descendants of a hidden unit (and have thus implicitly shown that as long as f has more than one perceptron, such decompositions exist). But such a choice of W is not

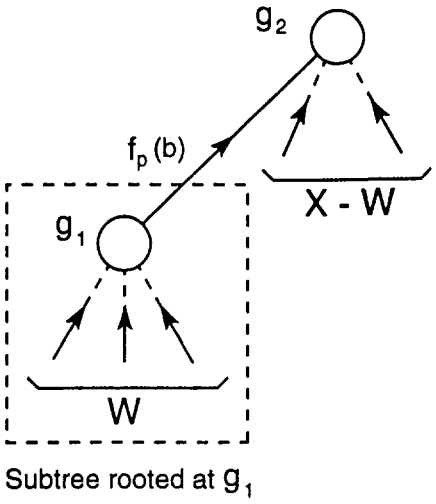

Figure 3. A partition of the variables X into $W \cup (X - W)$. The set of variables W are the descendants of node g_2. We may think of g_1 and g_2 as some nodes in our target net.

necessary for a decomposition to be valid, and we have no means to determine whether a given valid decomposition has any architectural significance in f. The key fact is that from *any* valid decomposition we shall be able to find a consistent NPN by forming a subnetwork over the W variables, regardless of whether the resulting architecture agrees with f's. Thus while the converse of Lemma 5 does not hold, it almost holds, in the sense that if a decomposition is valid on M then there is a NPN that agrees with f on M, in which the W variables appear in a subnetwork computing f_p. (This potential use of decompositions that are not partitions of X according to subformula of f is a key difference from other read-once formula algorithms.)

Generating a valid decomposition is the subject of the next section. Here we present the logic for the divide and conquer approach by which we build the network given decompositions. Lemma 6 below proves that this is correct. Besides the variable set X and the sample M, this routine also uses a membership oracle for f.

LEMMA 6. *Suppose f is a NPN over X and M is a set of examples classified according to f. Suppose there is a polynomial time algorithm that produces a set of decompositions, one of which is valid and non-trivial (provided a valid non-trivial decomposition for M exists). Then there is a polynomial time algorithm to find a NPN consistent with f on M.*

Proof: We prove that *Find-Consistent-NPN* (figure 4) is such a routine. This is mostly a consequence of previous observations, with a few additional points. First note that valid non-trivial decompositions can fail to exist only in the case where f has a single computation unit, so by assumption we fail to find one only when linear programming can indeed produce a consistent hypothesis.

Find-Consistent-NPN(X, M)

1. Apply the reductions of Section 4 to find justifying assignments for each variable, normalize the sample, and reduce the problem to the case where f is monotone.
2. Invoke the next section's routine *Find-Decomposition* to generate a valid non-trivial decomposition (x_i, a, W). If none is found, train a single perceptron to fit M using linear programming, and return.
3. Invoke *Find-Consistent-NPN* with the following inputs
   - Variable set W.
   - The sample obtained by M by taking each example $b \in M$ and deleting the variable settings for $X - W$.
   - A (simulated) membership query oracle for the projection f_p where p is the partial assignment $a_{W \leftarrow *}$.

   Let h_1 be the NPN returned.
4. Invoke *Find-Consistent-NPN* with the following inputs
   - Variable set $(X - W) \cup \{x_i\}$.
   - The sample obtained by M by taking each example $b \in M$ and deleting the variable settings for $X - \{x_i\}$ and resetting x_i to $f_p(b)$ (either 0 or 1).
   - A (simulated) membership query oracle for the projection f_q where q is the partial assignment $a_{(X-W)\cup\{x_i\} \leftarrow *}$.

   Let h_2 be the NPN returned.
5. Return the hypothesis h obtained by substituting h_1 for the x_i input to h_2.

Figure 4. Routine *Find-Consistent-NPN*.

Assume that the two recursive calls produce NPNs h_1 over W and h_2 over $(X-W)\cup\{x_i\}$ that are consistent with f_p and f_q on their respective samples. By construction, our final hypothesis will be a NPN h over X, such that for any $b \in M$

$$\begin{aligned} h(b) &= h_2(b_{x_i \leftarrow h_1(b)}) \\ &= h_2(b_{x_i \leftarrow f_p(b)}) \\ &= f_q(b_{x_i \leftarrow f_p(b)}) \\ &= f(b_{W-\{x_i\} \leftarrow a, x_i \leftarrow f_p(b)}) \\ &= f(b). \end{aligned}$$

(The second equality comes from the correctness of h_1, the third from the correctness of h_2, the fourth from the definition of q, and the fifth from the definition of a valid of decomposition.)

Note that both f_p and f_q are projections of f and hence are realizable as NPNs for any choice of variables W (recall that the class of NPNs is projection closed). Hence it does not matter if the set W does not include the variables from some subnetwork of f, as long as the decomposition is valid. The recursive calls are indeed valid NPN learning problems, and we can prove correctness by induction. The induction is over the sum of the number

of variables in X plus the number of non-boolean variables in X. Since the decomposition is non-trivial, W always contains fewer variables than X (and no additional non-boolean variables). The only case in which $(X - W) \cup \{x_i\}$ contains as many variables as X is when x_i is non-boolean in M, but then x_i becomes boolean for the recursion (taking on the value of f_p), so the number of non-boolean variables decreases.

To bound the number of recursive calls, first note that we can only recurse n times for a decomposition with $w = |W| = 1$. This is because such a decomposition is used only when the single variable x_i in W is non-boolean, and each non-boolean input variable can be separated off in this manner only once (this decomposition corresponds to creating a unary perceptron whose only input is x_i, and hence we are in effect replacing the non-boolean x_i by the boolean output of this gate). The number of times we can recurse using a non-trivial decomposition in which $w = |W| > 1$ is bounded by $T(n)$, as specified by the following recurrence (i.e. we break a problem on n variables into separate problems on w and $n - w + 1$ variables):

$$\begin{aligned} T(n) &= 1 + \max_{1<w<n} (T(n - w + 1) + T(w)) \\ T(2) &= 0. \end{aligned}$$

It is easily verified that $T(n) = n - 2$. Thus this recursive processing can result in at most $2n - 2$ invocations of *Find-Consistent-NPN* (and of that quantity over half are for a case where the target is a simple threshold of a single non-boolean variable). ■

6.2. *Finding valid decompositions*

In this subsection we solve our main technical problem of finding valid decompositions. The decompositions we look for are ones obtained by starting with the most influential input variable x_i for some bottom level unit, and then calculating the set W of siblings of x_i (actually the techniques do not require that x_i's parent be bottom level, but it makes the arguments slightly cleaner). Of course we do not know in advance whether variable x_i is the most influential input to its parent. But it suffices to try our techniques for each possible x_i in succession, noting that when we get to a "good" x_i we shall indeed generate a valid non-trivial decomposition. It is conceivable that a "bad" x_i will still somehow let us generate a valid non-trivial decomposition, but that presents no difficulty since any such decomposition works for our divide and conquer approach. (Recall that with membership queries we can easily verify whether a proposed decomposition is valid.)

Suppose for now (and most of the remainder of the section) that we have prior knowledge that x_i is the most influential input to a bottom level unit. Let $g(x_i, z_1, \ldots, z_r)$ be the linear threshold function computed at x_i's parent. Let us define the following two conditions (expressing whether the single most influential input has enough power to overrule the aggregation of all other inputs):

$$g(1, 0, \ldots, 0) = 0 \tag{3}$$

$$g(0, 1, \ldots, 1) = 1. \tag{4}$$

Find-Decomposition(X, M)

1. Repeat for each $x_i \in X$ (until a valid non-trivial decomposition is found).
   a. Let (x_i, a, W) be the decomposition returned by invoking *Find-Decomp-1*(X,M,x_i). If (x_i, a, W) is valid and non-trivial, return this decomposition
   b. Let (x_i, a, W) be the decomposition returned by invoking *Find-Decomp-2*(X,M,x_i). If (x_i, a, W) is valid and non-trivial, return this decomposition
2. Return "failure" (the target NPN has no valid non-trivial decompositions).

Figure 5. Subroutine *Find-Decomposition.*

We use two different routines that find a valid decomposition: One is for the case where both conditions (3) and (4) hold (which is shown to suffice for a boolean x_i), and the other for the case where one (or both) of the conditions is false. Those routines (*Find-Decomp-1* and *Find-Decomp-2* respectively) are described in subsequent sections. Their correctness will suffice to show that we can generate a valid non-trivial decomposition by the routine *Find-Decomposition* shown in figure 5.

Before presenting the two routines to find decompositions, we prove the following lemma that gives a criterion useful for deciding whether a variable x_j is a sibling of x_i.

LEMMA 7. *Suppose f is a NPN over X in which variables x_i and x_j are siblings and in which x_i is at least as influential an input to their parent as is x_j. If p is a partial assignment that assigns values to $X - \{x_i, x_j\}$ such that f_p depends on x_j for values of x_i and x_j in $[0, 1]$, then f_p must also depend on x_i for values of x_i and x_j in that range.*

Proof: Since f_p depends on x_j, there is some value $x^{(0)} \in [0, 1]$ for x_i on which

$$f(p_{x_i \leftarrow x^{(0)}, x_j \leftarrow 0}) = 0$$
$$f(p_{x_i \leftarrow x^{(0)}, x_j \leftarrow 1}) = 1$$

For f_p not to depend on x_i it must be true then that

$$f(p_{x_i \leftarrow 1, x_j \leftarrow 0}) = 0 \tag{5}$$
$$f(p_{x_i \leftarrow 0, x_j \leftarrow 1}) = 1 \tag{6}$$

But this leads to a contradiction, since the fact that x_i is at least as influential as x_j implies that $p_{x_i \leftarrow 1, x_j \leftarrow 0}$ has at least as high a weighted sum of inputs to x_i's parent as does $p_{x_i \leftarrow 0, x_j \leftarrow 1}$. Since the assignments agree elsewhere, monotonicity yields the contradiction that condition (5) cannot hold unless condition (6) is false. ■

6.2.1. The boolean case

In this section we show how to find a decomposition in the case where x_i (the most influential input to a bottom level unit) is boolean. This technique will also work for a non-boolean x_i whose parent computes a function g satisfying both conditions (3) and (4) listed above.

OR-Sibling-Test(x_i, x_j, X, p)

1. Repeat the following for each $x_k \in X - \{x_i, x_j\}$,
   a. Let p' be $p_{x_k \leftarrow 0}$.
   b. If $f_{p'}$ also computes the function "$(x_i = 1)$ OR $(x_j = 1)$" on $\{0,1\} \times \{0,1\}$, then reset p to p'.
   c. Otherwise if $f_{p'}$ computes the function "$x_j = 1$", return "Not siblings".
2. Return "Are siblings".

Figure 6. Subroutine *OR-Sibling-Test.*

First we present a key subroutine, *OR-Sibling-Test.* This subroutine takes as input a projection p that depends on the two variables x_i and x_j and that computes "$(x_i = 1)$ OR $(x_j = 1)$" when evaluated on the four input settings from $\{0,1\} \times \{0,1\}$. The technique is to set other variables to 0 in p, one at a time, trying to make the projection depend on just x_j. This is not possible (by Lemma 7) if x_i and x_j are siblings and x_i has the higher weight. We shall show, however, that this is possible if x_i and x_j are not siblings and if x_i's parent is a bottom level unit for which condition (3) holds. This gives us a means to test whether another variable x_j is a descendant of x_i's parent under the assumption that x_i is the most influential input. The remaining processing for this case will involve finding a suitable projection p with which to invoke *OR-Sibling-Test* (figure 6).

LEMMA 8. *Suppose f is a NPN over X and x_i is the most influential input to a bottom level unit. Suppose $x_j \in X$ and p is a partial assignment assigning values to $X - \{x_i, x_j\}$ such that on the domain $\{0,1\} \times \{0,1\}$, f_p computes the function "$(x_i = 1)$ OR $(x_j = 1)$." Then if x_i's parent computes a function $g(x_i, z_1, \ldots, z_r)$ satisfying $g(1, 0, \ldots, 0) = 0$, the routine OR-Sibling-Test(x_i, x_j, X, p) will return "Are siblings" if and only if x_i and x_j are siblings in f.*

Proof: The routine returns "Not Siblings" only in a case where $f_{p'}$ depends on x_j, but not x_i. By Lemma 7 this cannot occur unless x_i and x_j are indeed not siblings. Now suppose x_j is not a sibling of x_i, and x_i's parent computes a function g of the indicated form. Each change to a variable x_k in the main loop, preserves the property that g_p outputs the value of x_i (for $x_i \in \{0,1\}$). We cannot set all the siblings of x_i to 0 in p, since by assumption we would then have $g_p \equiv 0$. So on some iteration we must have some sibling x_k that we cannot set to 0 in p, meaning that $g_{p'} \equiv 0$. In this case $f_{p'}$ will compute "$x_j = 1$", and we shall correctly return "Not Siblings". ■

We will also make use of a subroutine *AND-Sibling-Test* that is the dual of the previous routine, obtained by changing "OR" to "AND" and "$p_{x_k \leftarrow 0}$" to "$p_{x_k \leftarrow 1}$". This subroutine will take a projection equivalent to "$(x_i = 1)$ AND $(x_j = 1)$" and, given the same conditions as Lemma 8 excepting that g satisfies condition (4) rather than (3), will return "Are Siblings" if and only if x_j is a sibling of x_i in f.

Find-Decomp-1(X, M, x_i)

1. Initialize W to $\{x_i\}$.
2. For each variable $x_j \in X - \{x_i\}$,
   a. Pick a justifying assignment $a \in M$ for x_j (i.e. $f(a_{x_j \leftarrow 1}) \neq f(a_{x_j \leftarrow 0})$).
   b. Evaluate $f_{a_{x_i, x_j \leftarrow *}}$ on the four settings of x_i, x_j from $\{0,1\} \times \{0,1\}$.
   c. If on this domain, that projection is equivalent to "$(x_i = 1)$ OR $(x_j = 1)$", invoke

      OR-Sibling-Test$(x_i, x_j, X, a_{x_i, x_j \leftarrow *})$.

      Add x_j to W if that subroutine returns "Are siblings".
   d. Otherwise, if the projection is equivalent to "$(x_i = 1)$ AND $(x_j = 1)$", invoke

      AND-Sibling-Test$(x_i, x_j, X, a_{x_i, x_j \leftarrow *})$.

      Add x_j to W if that subroutine returns "Are siblings".
3. Pick an $a \in M$ that is a justifying assignment for x_i, and return the decomposition (x_i, a, W).

Figure 7. Subroutine *Find-Decomp-1*.

Before presenting our routine to find a decomposition when conditions (3) and (4) both hold, we argue that this assumption is (almost) without loss of generality for a boolean x_i. Suppose x_i is the most influential input to $g(x_i, z_1, \ldots, z_r)$. We ignore without loss of generality the possibility that $r = 0$. In that case the unary function g (if non-constant) could compute just the identity function on its boolean input, and we may assume there are no such useless units in f. If g fails condition (3) (i.e. $g(1, 0, \ldots, 0) = 1$), then it is true from monotonicity that g is equivalent to

$$(x_i = 1) \text{ OR } g(0, z_1, \ldots, z_r).$$

If $r > 1$, then this means f can be expressed with more hidden units by splitting g in this manner. If we assume (without loss of generality) that f is expressed so that there are as may hidden units as possible, it follows that condition (3) must be true for a boolean variable x_i (unless its parent has only two inputs, i.e. $r = 1$). A similar argument shows that condition (4) must also hold for boolean variables (if $r > 1$). We can handle the $r = 1$ case by simply testing all pairs of variables to see if they can form (along with a justifying assignment for one of them) a valid non-trivial decomposition.

The routine *Find-Decomp-1* (figure 7) will find a valid decomposition if both conditions (3) and (4) hold. This, along with a test of each pair of variables to cover the $r = 1$ case, is sufficient to find a valid non-trivial decomposition if x_i is boolean (in fact we don't include these tests of pairs of variables in our routine, since the case where condition (3) or condition (4) fails is covered in the next section by the processing necessary for non-boolean variables).

LEMMA 9. *Suppose f is a NPN over X and M is a sample of examples classified according to f. Suppose that x_i is the most influential variable connected to some bottom level unit.*

Further suppose that the following two statements hold for the linear threshold function $g(x_i, z_1, \ldots, z_r)$ computed at x_i's parent:

$$g(1, 0, \ldots, 0) = 0$$
$$g(0, 1, \ldots, 1) = 1.$$

Then (if f has more than one computation unit) Find-Decomp-1(X, M, x_i) will return a valid non-trivial decomposition for M.

Proof: We shall show that W is set to exactly those variables that are children of x_i's parent. The claim then follows from Lemma 5 as long as f has more than one perceptron (to guarantee non-triviality).

Suppose x_j is a sibling of x_i. It follows from Lemma 7 that $f_{a_{x_i, x_j \leftarrow *}}$ must depend on both x_i and x_j. Since we consider only the values 0 and 1 for each variable, the only two possible functions are AND and OR. Thus we call either *OR-Sibling-Test* or *AND-Sibling-Test*. The conditions on g and x_i guarantee that in either case we shall add x_j to W if and only if it is a sibling of x_i. ■

6.2.2. *The non-boolean case*

Suppose the most influential input, x_i, to some bottom level unit is non-boolean. In this case our previous routine *Find-Decomp-1* will still work if both conditions (3) and (4) hold, but we can no longer ignore the possibility that one of the two does not.

In this section we present a second routine (*Find-Decomp-2* in figure 8) that will find a valid decomposition in such cases. The routine incrementally builds a set W of proposed siblings for x_i. This set starts simply as $W = \{x_i\}$, and is augmented only by variables guaranteed to be x_i's siblings (given our assumptions on g and that x_i is the most influential input to a bottom level unit). If W grows to contain all x_i's siblings, then of course the decomposition (x_i, a, W) will be valid (for a a justifying assignment for x_i). If for some partial W the decomposition is not valid, it is because some $b \in M$ fails criterion (2). We show how to manipulate this b (along with a) to find a new sibling of x_i. Repeating this will eventually lead us to a valid decomposition.

The following lemma gives us a test useful to guarantee that a new variable x_j is indeed a sibling of x_i.

LEMMA 10. *Suppose f is a NPN over X, and $x_i \in X$ is a variable that is the most influential input to a bottom level unit. Suppose $W \subset X$ includes only x_i and siblings of x_i. Let q_1 and q_2 be two partial assignments that assign values to $X - W$ and differ only on the value they assign to some single variable $x_j \in X - W$. If neither f_{q_1} nor f_{q_2} is constant and if $f_{q_1} \not\equiv f_{q_2}$, then x_j is a sibling of x_i in f.*

Proof: Let g be the weighted threshold function computed at x_i's parent. Since f_{q_1} and f_{q_2} depend only on inputs to g, it follows (from Lemma 4) that $f_{q_1} \equiv g_{q_1}$ and $f_{q_2} \equiv g_{q_2}$. Hence $g_{q_1} \not\equiv g_{q_2}$, which is possible only if x_j is an input to g. ■

Find-Decomp-2(X, M, x_i)

1. Initialize W to $\{x_i\}$.
2. Let $a \in M$ be a justifying assignment for x_i (i.e. $f(a_{x_i \leftarrow 1}) \neq f(a_{x_i \leftarrow 0})$).
3. Repeat the following until either (x_i, a, W) is a valid decomposition for M, or until W stabilizes.
   a. Let $b \in M$ be an assignment for which the decomposition is not valid (i.e. for $p = a_{W \leftarrow *}$,
   $$f(b) \neq f(b_{W - \{x_i\} \leftarrow a, x_i \leftarrow f_p(b)}).)$$
   b. Define $q = b_{W \leftarrow *}$. Let c be whichever of b or $b_{W - \{x_i\} \leftarrow a, x_i \leftarrow f_p(b)}$ has $f_p(c) \neq f_q(c)$ (one must).
   c. Repeat for each variable $x_j \in X - W$ that p and q assign differently (i.e. $p_j \neq q_j$).
      i. Let $p' = p_{x_j \leftarrow q_j}$. If $f_{p'}$ is non-constant (as determined by checking whether it evaluates to 1 on the all 1's setting and 0 on the all 0's setting),
         A. If $f_{p'}(c) \neq f_p(c)$, add x_j to W and go back to the start of the step 3 loop.
         B. Otherwise ($f_{p'}(c) = f_p(c)$), reset p to p' and repeat step 3c.
      ii. Else, let $q' = q_{x_j \leftarrow p_j}$. If $f_{q'}$ is non-constant,
         A. If $f_{q'}(c) \neq f_q(c)$, add x_j to W and go back to the start of the step 3 loop.
         B. Otherwise ($f_{q'}(c) = f_q(c)$), reset q to q' and repeat step 3c.
4. Return (x_i, a, W).

Figure 8. Subroutine *Find-Decomp-2*.

Now we present the partitioning routine, followed by a proof its correctness. The basic idea is that if our decomposition (x_i, a, W) is not valid on an example b, then it must be the case that f_p and f_q (where $p = a_{W \leftarrow *}$ and $q = b_{W \leftarrow *}$) are different non-constant projections. We try to move p and q towards each other by changing variables on which they disagree (while preserving $f_p \not\equiv f_q$), and argue that when we get stuck we can apply Lemma 10 to add a new variable to W. See figure 8.

LEMMA 11. *Suppose f is a NPN over X and M is a sample of examples classified according to f. Suppose non-boolean variable x_i is the most influential input connected to some bottom level unit. Further suppose the linear threshold function $g(x_i, z_1, \ldots, z_r)$ computed at x_i's parent satisfies at least one of the following two conditions:*

$$g(1, 0, \ldots, 0) = 1$$
$$g(0, 1, \ldots, 1) = 0.$$

Then Find-Decomp-2(X, M, x_i) will return a valid non-trivial decomposition for M.

Proof: Whenever our decomposition (x_i, a, W) is not valid for M it is because (by definition) M contains an example b as selected at step 3a. Let $b' = b_{W - \{x_i\} \leftarrow a, x_i \leftarrow f_p(b)}$ for this b. Since b and b' differ only on variables in W, if follows that for $q = b_{W \leftarrow *}$,

$$f_q(b) = f(b) \neq f(b') = f_q(b').$$

However by definition of p as $a_{W \leftarrow *}$ where a is a justifying assignment for a it follows that

$$f_p(b') = f(p/b') = f(a_{W \leftarrow b'}) = f(a_{x_i \leftarrow f_p(b)}) = f_p(b).$$

Thus as claimed at step 3b, f_q and f_p must disagree on either b or b' (henceforth called c).

We already know that f_p is non-constant, since p is obtained from a justifying assignment for $x_i \in W$. As observed above $f_q(b) \neq f_q(b')$, implying that f_q is also non-constant. Note that these properties (along with $f_q(c) \neq f_p(c)$) are preserved by the changes to p and q during the step 3c loop.

Thus it is easy to prove by induction that the set W includes only x_i and siblings of x_i. This is true initially for $W = \{x_i\}$, and each time we add a variable x_j to W within step 3c the condition of Lemma 10 implies that x_j must indeed be a sibling of x_i (we have observed that both f_p and f_q are non-constant). The decomposition (x_i, a, W) will be valid if W contains all of x_i's siblings, so the remaining point we must prove is that on each iteration of step 3 we shall indeed add a new sibling to W.

Consider the processing of that step 3c loop. The fact that $f_p(c) \neq f_q(c)$ implies that p and q must always disagree on at least one sibling of x_i (since p and q leave unassigned only inputs to g, it would be a contradiction for them to induce the same projection on that function). We shall prove that each time the x_j we consider in the loop is indeed a sibling of x_i, we either change p and q to agree on the variable or (as hoped) we add x_j to W. Since we cannot do the former for every such x_j while preserving $f_p \not\equiv f_q$, we must eventually do the latter for one.

To prove this final claim, it is enough to show that the stated restriction on g implies that at least one of the two projections $f_{p'}$ and $f_{q'}$ considered in the loop must be non-constant, assuming x_j is a sibling of x_i. Suppose, without loss of generality, that on entering the loop $p_j > q_j$. Then if $f_{p'}$ is constant (recall $p' = p_{x_j \leftarrow q_j}$) while f_p was not, monotonicity implies $f_{p'} \equiv 0$. Similarly if $f_{q'}$ is constant, $f_{q'} \equiv 1$. Neither p' nor q' assigns a value to x_i, so $f_{p'} \equiv 0$ is possible only if $g(1, 0, \ldots, 0) = 0$, and $f_{q'} \equiv 1$ is possible only if $g(0, 1, \ldots, 1) = 1$. It contradicts the lemma for both these statements to hold. ■

7. Main theorem and analysis

In this section we put together the pieces of our algorithm, along with some straightforward analysis, to prove the following main theorem. First, however, we must somehow address the computational complexity issues of bit precision in our examples. The algorithm we have presented performs essentially just symbolic manipulations, and a unit cost model for manipulating numbers seems the most natural. But the linear programming subroutine will take time proportional to the number of bits in the problem statement, and hence an adversary can blow up the running time by specifying a probability distribution in which the examples are clustered very close to the separating half-plane. To avoid this we adopt essentially the approach of Baum (1991), and allow our algorithm time proportional to the number of bits of precision s appearing in our random examples. The number of bits in the linear programming problems will then be bounded by the product of s, n, and the sample size.

THEOREM 1. *The class of nonoverlapping perceptron networks over n inputs is PAC learnable using examples and membership queries in time polynomial in n and the number of bits in the examples. The algorithm uses*

$$m = \max\left(\frac{4}{\epsilon}\log\frac{2}{\delta}, \frac{104n\log(2en) + 32n\log\log(4n)}{\epsilon}\log\frac{13}{\epsilon}\right)$$

random examples, $O(n^2m)$ membership queries, and runs in time $O(n^{4.5}sm)$ (where s is the maximum number of bits of precision specified for the examples in the random sample).

Proof: By Corollary 1 it suffices to output a NPN consistent with a sample M of size m to achieve PAC learning. The algorithm is to draw such a sample and invoke *Find-Consistent-NPN*(X, M). By Lemma 6 this routine will output such a NPN, provided that *Find-Decomposition* finds a valid non-trivial decomposition whenever f contains more than one computation unit. Since we can check whether a given decomposition is valid and non-trivial using membership queries, the correctness of this routine follows from Lemmas 9 and 11.

Now we analyze the running time for an n variable NPN on a sample of size m. A single invocation of *OR-Sibling-Test* uses $O(n)$ time and queries. Thus an invocation of *Find-Decomp-1* takes $O(n^2)$ time and queries. A single invocation of routine *Find-Decomp-2* on n variables takes $O(nm + n^2)$ time and membership queries. We argued previously that there are only $O(n)$ recursive calls to *Find-Consistent-NPN*. We note that we can conserve processing by not repeating similar calls to *Find-Decomp-1* and *Find-Decomp-2* in deeper recursions. As we mentioned previously, those routines can be shown to succeed as long as x_i is the most influential input to its parent, regardless of whether that parent is a bottom level unit. Thus to find all the necessary decompositions in deeper levels of recursion we need only run the partitioning routine on the "new" variable we have introduced. Thus the total cost in partitioning can be reduced to $O(n^2m+n^3)$ due to real variables, and $O(n^3)$ for boolean variables (or a factor n more for the routines as written without the optimization).

The cost associated with the recursion itself and with testing whether decompositions are valid is $O(nm)$ total. The preliminary processing (eliminating irrelevant variables, reducing to the monotone case, etc.) is $O(n^2m)$. The final stage of the algorithm is to run linear programming on various partitions of the variables. The total cost of this is at most $O(n^{3.5}L)$, where L is the number of bits in the instance of the linear programming problem (Karmarkar, 1984), which is $O(snm)$. ■

It is interesting to note that the cost of linear programming dominates the running time.

8. Conclusion

This study was aimed at finding whether or not neural nets with no overlap between the receptive fields of their nodes are somewhat easier to learn. The answer to this question depends on the resources available to the learner. If only examples are available, the answer is no (based on cryptographic assumptions).

Assuming that membership queries are allowed, we presented an algorithm that PAC learns any nonoverlapping perceptron network. The algorithm works by breaking the learning problem down to its elementary components: learning *independent perceptrons*. The queries used by the algorithm are very simple and amount to asking what variable(s) is causing a given example to be positive (negative). To our knowledge, it is the only known result for learning perceptron networks where an algorithm is able to identify both the architecture and the weight values necessary to solve the learning problem. The generalization of the algorithm to nonoverlapping perceptron networks with more than one output is straightforward.

A number of problems remain open. One important problem is whether or not nonoverlapping perceptron networks are learnable from examples only under the uniform distribution. By analogy, Schapire (1991) has shown that some classes of read-once formulas are learnable under those conditions, and we have preliminary results for subclasses of NPN's (Golea, Marchand, and Hancock, 1992). Learning general (overlapping) perceptron networks on the uniform distribution remains intractable, given standard cryptographic assumptions (Kharitonov, 1993).

Acknowledgments

This research took place while Tom Hancock was a graduate student at Harvard University, supported by ONR grant N00014-85-K-0445 and NSF grant NSF-CCR-89-02500. Mostefa Golea and Mario Marchand are supported by NSERC grant OGP-0122405.

We thank Sleiman Matar, Les Valiant, and the anonymous referees for their helpful comments.

Notes

1. The Blum-Rivest and Judd results show intractability assuming P≠NP, while the representation independent results rely on stronger assumptions from cryptography, such as the difficulty of factoring.
2. We omit the description of this measure theoretic condition, which is easily satisfied by the class we consider.

References

Angluin, D., Hellerstein, L., and Karpinski, M. (1993). Learning read-once formulas with queries. *Journal of the Association for Computing Machinery,* 40, 185–210.

Angluin, D. and Kharitonov, M. (1991). When won't membership queries help? *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing* (pp. 444–454). New York: ACM Press.

Barkai, E., Hansel, D., and Kanter, I. (1990). Statistical mechanics of multilayer neural networks. *Physical Review Letters,* 65(18), 2312–2315.

Barkai, E. and Kanter, I. (1991). Storage capacity of a multilayer network with binary weights. *Europhysics Letters,* 14(2), 107–112.

Baum, E.B. (1990a). On learning a union of halfspaces. *Journal of Complexity,* 6, 67–101.

Baum, E.B. (1990b). A polynomial time algorithm that learns two hidden unit nets. *Neural Computation,* 2, 510–522.

Baum, E.B. (1991). Neural net algorithms that learn in polynomial time from examples and queries. *IEEE Transactions on Neural Networks,* 2, 5–19.

Baum, E.B. and Haussler, D. (1989). What size net gives valid generalization. *Neural Computation,* 1, 151–160.

Blum, A. and Rivest, R. (1988). Training a 3-node neural network is NP-complete. *Proceedings of the 1988 Workshop on Computational Learning Theory* (pp. 9–18). San Mateo, CA: Morgan Kaufman.

Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. (1989). Learnability and the Vapnik-Chervonenkis dimension. newblock *Journal of the Association for Computing Machinery,* 36(4), 929–965.

Bshouty, N.H., Hancock, T.R., and Hellerstein, L. (1992a). Learning arithmetic read-once formulas. *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing* (pp. 370–381). New York: ACM Press.

Bshouty, N.H., Hancock, T.R., and Hellerstein, L. (1992b). Learning boolean read-once formulas with arbitrary symmetric and constant fan-in gates. *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory* (pp. 1–15). New York: ACM Press.

Goldman, S., Kearns, M., and Schapire, R. (1990). Exact identification of circuits using fixed points of amplification functions. *Proceedings of the 31st Symposium on Foundations of Computer Science.* Los Alamitos, CA: IEEE Computer Society Press. To appear, *SIAM Journal of Computation.*

Golea, M., Marchand, M., and Hancock, T.R. (1992). On Learning μ-Perceptron Networks with Binary Weights. In S.J., Hanson; J., Cowan and C.L., Giles, (Eds), *Neural Information Processing Systems 5.* San Mateo, CA: Morgan Kaufman.

Hancock, T.R. (1991). Learning 2μ DNF formulas and $k\mu$ decision trees. *Proceedings of the Fourth Annual Workshop on Computational Learning Theory* (pp. 199–209). San Mateo, CA: Morgan Kaufman.

Judd, S. (1988). On the complexity of loading shallow neural networks. *Journal of Complexity,* 4, 177–192.

Karmarkar, N. (1984). A new polynomial time algorithm for linear programming *Combinatorica,* 4, 373–395.

Kearns, M., Li, M., Pitt, L., and Valiant, L. (1987). On the learnability of boolean formulae. *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing.* New York: ACM Press.

Kearns, M. and Valiant, L. (1989). Cryptographic limitations on learning boolean formulae and finite automata. *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing* (pp. 433–444). New York: ACM Press.

Kharitonov, M. (1993). When won't membership queries help? *Proceedings of the Twenty Fifth Annual ACM Symposium on Theory of Computing.* New York: ACM Press.

Lin, J.H. and Vitter, J.S. (1991). Complexity results on learning by neural nets. *Machine Learning,* 6, 211–230.

Pagallo, G. and Haussler, D. (1989). *A greedy method for learning μDNF functions under the uniform distribution* (Technical Report UCSC-CRL-89-12). Santa Cruz, CA: Deptartment of Computer and Information Science, University of California at Santa Cruz.

Schapire, R.E. (1991). Learning probabilistic read-once formulas on product distributions. *Proceedings of the Fourth Annual Workshop on Computational Learning Theory* (pp. 184–198). San Mateo, CA: Morgan Kaufman.

Valiant, L.G. (1984). A theory of the learnable. *Communications of the ACM,* 27, 1134–1142.

Received November 9, 1991
Final Manuscript June 16, 1993