Network 4 (1993) 67–85. Printed in the UK

On learning simple neural concepts: from halfspace intersections to neural decision lists

Mario Marchand and Mostefa Golea
Ottawa-Carleton Institute for Physics, University of Ottawa, Ottawa, Canada K1N 6N5

Received 19 August 1991, in final form 2 June 1992

Abstract. In this paper, we take a close look at the problem of learning simple neural concepts under the uniform distribution of examples. By simple neural concepts we mean concepts that can be represented as simple combinations of perceptrons (halfspaces). One such class of concepts is the class of halfspace intersections. By formalizing the problem of learning halfspace intersections as a *set-covering problem*, we are led to consider the following sub-problem: given a set of nonlinearly separable examples, find the largest linearly separable subset of it. We give an approximation algorithm for this NP-hard sub-problem. Simulations, on both linearly and nonlinearly separable functions, show that this approximation algorithm works well under the uniform distribution, outperforming the pocket algorithm used by many constructive neural algorithms. Based on this approximation algorithm, we present a greedy method for learning halfspace intersections. We also present extensive numerical results that strongly suggest that this greedy method learns halfspace intersections under the uniform distribution of examples. Finally, we introduce a new class of simple, yet very rich, neural concepts that we call *neural decision lists*. We show how the greedy method can be generalized to handle this class of concepts. Both greedy methods for halfspace intersections and neural decision lists were tried on real-world data with very encouraging results. This shows that these concepts are not only important from the theoretical point of view, but also in practice.

1. Introduction

Learning in feedforward layered neural networks has attracted much attention recently [39]. Unfortunately, training these systems has certainly proven to be a very difficult task. It is now recognized that the most popular 'learning rule', backpropagation [38], generally needs prohibitive training times because of the local minimum problem. This has culminated in the important work of Judd [25] who showed the NP-hardness of the problem of training in *fixed* network architectures. To circumvent this problem, several constructive (or growth) algorithms have been proposed recently [13–15, 18, 28–30, 37, 41, 43]. These algorithms share the feature that the network architecture is not fixed (and guessed) before training. Instead, units are added, one by one, by using an algorithm that minimizes some error criterion at the single-neuron level. As a consequence, these algorithms generally run much faster than backpropagation. The upshot is that one can (and generally does) assist a spectacular explosion in the number of neurons needed to load the data. In these situations, the network simply acts as a table lookup and exhibits no generalization. As a result, very few good generalization results have been reported for these constructive algorithms.

A meaningful question to ask here is whether or not one can learn concepts (functions) representable as relatively simple feedforward nets (FFNs). To be able to answer this question, we need to define more precisely what we mean by learning. For that, we appeal to the PAC learning model [7, 21, 32, 44]. Loosely speaking, the learning algorithm has access

0954-898X/93/010067+19$07.50

to a set of examples generated according to a fixed but otherwise arbitrary distribution $P(x)$. The examples are labelled according to an unknown target function which may be any one from some known class of functions. The algorithm is said to be efficient if, given a 'reasonable' number of examples, it is 'likely' to produce, in polynomial time, a 'good' approximation of the unknown function. For precise definitions, the reader is referred to the literature cited.

If we adopt the PAC's point of view, the list of neural networks that are learnable is deceptively small [6,27]. Even simple neural concepts like halfspace intersections are known to be not properly learnable under an arbitrary distribution of examples. To our knowledge, only single halfspaces [7] and border-augmented symmetric differences of halfspaces [45] have proven to be learnable.

In view of the scarcity of positive PAC learning results, researchers have looked for positive results by providing the learning algorithm with additional information in the form of queries [1,20] or by restricting the distribution of examples [5,17]. Taking the second approach, Bartlett and Williamson [2] have proposed to permit only *reasonable* distributions (i.e. bounded distributions which are non-zero everywhere in the domain). One such distribution is the uniform one.

In this paper, we first investigate the learnability of halfspace intersections under the uniform distribution of examples. Our task is to find, in polynomial time, a halfspace intersection hypothesis net that approximates the most to the halfspace intersection target net. Furthermore, we allow for a possibly larger number of halfspaces in the hypothesis net. By the Occam's razor principle [7 8], if the hypothesis function is not too large compared with the target function, we are guaranteed to learn. We stress here the fact that algorithms such as the backpropagation [38] and the cascade-correlation [13] do not solve our problem because there is no guarantee that they do converge to a solution in polynomial time. We present a greedy method for this problem which, although we are not yet able to prove its PAC correctness, does very well experimentally under the uniform distribution of examples up to 50 dimensions. To our knowledge, the tests reported here go beyond any in the literature in terms of testing generalization by a greedy method in high dimensions.

The greedy method, like all the other constructive algorithms, is built around a single-perceptron training procedure. This procedure tries to find a halfspace consistent with all the positive examples and a large number of negative examples. Because finding the halfspace consistent with the *largest* subset of negative examples is NP-hard, we give an approximation algorithm for it. In view of the fact that it is a linear programming (LP) problem to find whether or not a data set is linearly separable—and for which there exist very efficient algorithms—our approximation algorithm incorporates an 'incremental' LP algorithm (IncLP). We present numerical evidence for the superiority of this single-perceptron training procedure over the pocket algorithm, used by many constructive neural algorithms.

The greedy method for halfspace intersections is extended to a class of functions we call *neural decision lists*. These are a generalization of the decision lists of Rivest [35] by allowing each node to be a halfspace (perceptron). This class of functions is strictly richer than halfspace intersections (unions).

Both greedy methods for halfspace intersections and neural decision lists are tried on real-world data with very encouraging results. Their performance is comparable to C4, a 'state of the art' tree-induction algorithm [34]. This shows that these simple neural concepts are not only important from the theoretical point of view, but also in practice.

This paper is organized as follows. In section 2 we present some definitions. Our greedy method for halfspace intersections is presented in section 3. In section 4, we present our

approximation algorithm for training single neurons. Our approach is generalized to neural decision lists in section 5. In section 6, we present the numerical results of our extensive simulation on both random nets and real-world data. The conclusions are summarized in section 7.

2. Definitions

Let $X = \{x_1, x_2, \ldots, x_n\}$ be the set of n-input variables and I^n the n-dimensional input (or instance) space which can either be $\{-1, +1\}^n$, $[-1, +1]^n$ or a subset of $\mathcal{R}^n$.

An *example* of a boolean function $f : I^n \to \{-1, +1\}$, is an an ordered pair $\langle \boldsymbol{x}, f(\boldsymbol{x}) \rangle$, where $\boldsymbol{x} \in I^n$. Point $\boldsymbol{x}$ is said to be a positive example if $f(\boldsymbol{x}) = +1$, otherwise it is said to be a negative example. A *sample* is a set of examples. We assume that the distribution D generating the examples is uniform on I^n.

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 bias w_0. The output of the function is $+1$ or -1 depending on whether the following inequality holds:

$$\sum_{x_i \in X} w_i x_i + w_0 > 0 .$$

Such functions are also referred to as *perceptrons* or *halfspaces*. We denote by H the positive halfspace $\{\boldsymbol{x} : \boldsymbol{w} \cdot \boldsymbol{x} + w_0 > 0\}$ and by $\overline{H}$ its complement. Halfspace H is said to *cover* example $\boldsymbol{x}$ if $\boldsymbol{x} \in H$. Halfspace H is said to be *consistent* with sample S if all positive examples of S are covered by H and and all negative examples of S are covered by $\overline{H}$.

A function $f : I^n \to \{-1, +1\}$ is said to be a *halfspace intersection* if it can be written as a conjunction (AND) of halfspaces. These functions have an obvious neural net representation: a FFN made of one layer of hidden units connected to a single output unit that performs the 'AND' operation (see figure 1).

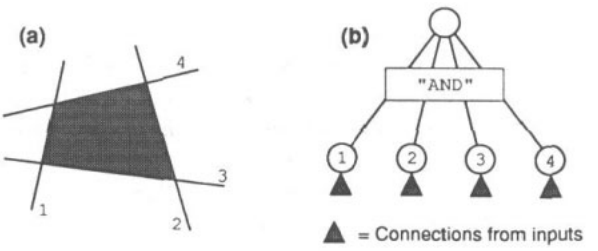

Figure 1. (*a*) The function represented by the intersection of four halfspaces. The shaded region represents the set of positive examples. (*b*) The equivalent FFN. The input units are not shown.

We generalize the notion of *decision lists*, introduced by Rivest [35], to *neural decision lists* (NDL). A NDL (figure 2) is a list $\mathcal{L}$ of pairs

$$(H_1, v_1), (H_2, v_2), \ldots, (H_r, v_r)$$

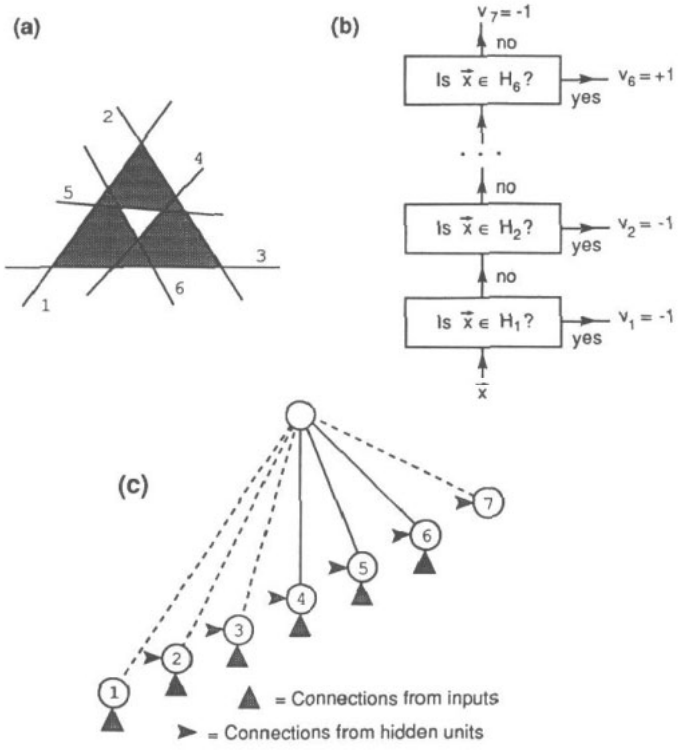

Figure 2. (*a*) A function to learn; the shaded region represents the set of positive examples. (*b*) A NDL performing this function. (*c*) The equivalent cascade FFN. Each hidden unit receives connections from the input units (not shown) and only from the other hidden units below it. Excitatory connections going to the output are indicated by a full line whereas inhibitory connections are shown by dashed lines.

where each H_i is a halfspace and v_i is a value in $\{-1, +1\}$. The last halfspace H_r is the constant function† $+1$. This defines a function as follows: for any $\boldsymbol{x}$, $\mathcal{L}(\boldsymbol{x})$ is defined to be equal to v_j where j is the first (least) index for which $\boldsymbol{x} \in H_j$. As in [35], we may think of a NDL as an extended '`if-then-elseif-...else-`' rule (see figure 2). Compared to Rivest's decision lists, NDLs have the same structure, but the complexity of the decision allowed at each node is greater.

This class of representations is strictly richer than halfspace intersections (unions). Indeed, any boolean function on a boolean (or discrete) domain has a NDL representation. Moreover, there always exists a NDL consistent with any finite sample of a boolean function on a continuous domain (i.e. a subset of $\mathcal{R}^n$). We will present numerical results on 'real-world' data sets that have an efficient NDL representation.

NDLs have a simple FFN representation (see figure 2): it is a type of FFN known as a *cascade net* [13, 15, 27, 30] because hidden units need to be updated one after the other (in 'cascade'), starting from the first. Indeed, whenever an input $\boldsymbol{x}$ lies in the positive halfspace of the first hidden unit, this unit must decide its target. If it lies in the negative halfspace,

† We may think of the constant halfspace (function) as the halfspace covering the whole input space. A perceptron with zero weights and a positive bias will do the trick.

the decision must be postponed to the second hidden unit. If x lies in the positive halfspace of the second unit, this unit must decide its target. If not, the decision is postponed to the third hidden unit and so on until we find a halfspace H such that $x \in H$.

One way to simulate this hierarchy is to put strong inhibitory connections between the hidden units in such a way that, whenever an input vector x is presented to the net, we get an internal representation of the type $-1, -1, \ldots, -1, +1, -1, \ldots, -1$, i.e. each hidden unit outputs -1 except the one hidden unit which will actually decide the FFN output for x. In appendix A we show how to choose the inter-hidden-unit connections such that the equivalent FFN gives the same classification as the NDL.

3. Learning halfspace intersections

When given a sample from a halfspace intersection, the goal of the learner is to find a FFN that best approximates the target function. Here, each positive example must be consistent with each and every halfspace whereas each negative example only needs to be consistent with one halfspace. The problem, often called the credit assignment problem (CAP) [3], is to decide which halfspace must be consistent with a given negative example. One way to bypass the CAP is to adopt the following greedy method:

1. Let S^- be the set of all negative examples.
2. If S^- is empty, Halt.
3. Find the halfspace H that is consistent with the largest subset of S^- and all the positive examples.
4. Add H to the hypothesis net. Remove this subset from S^-. Go to 2.

Obviously, this greedy method will build as a hypothesis a halfspace intersection consistent with all the training examples.

Let us call the optimization problem encountered in step 3 the *densest-halfspace-covering problem*. Suppose for a moment that we have a way to solve this optimization problem *exactly*. It is easy to imagine some distributions of examples for which the halfspace that covers the largest number of negative examples is quite different from any one of the target halfspaces—thus causing the above greedy method to give a larger number of halfspaces than the minimum. This is not a real setback because our goal is not to find the *minimum* number of halfspaces, but to find a *good* approximation. Under the above assumption (that step 3 can be solved exactly), the greedy method is equivalent to the standard greedy algorithm for the set covering [9, 23]. Hence, it is guaranteed to find a halfspace intersection with a number of halfspaces not greater than $h \ln(m) + 1$ in the worst case, where m is the number of examples and h is the smallest number of halfspaces in any halfspace intersection consistent with all the examples. By the Occam's razor principle [7, 8], this would be sufficient to PAC-learn this class of concepts.

Unfortunately, the densest-halfspace-covering problem contains, as a particular case, a known NP-complete problem: the densest-hemisphere problem [16, 24] is finding the largest linearly separable subset (positive or negative) from a data set. Hence, for the greedy method to PAC-learn the class of halfspace intersections, one needs to find some approximation algorithm [16] for the densest-halfspace-covering problem with a good performance guarantee in the *worst case*. Unfortunately, we are not aware at present of any such algorithm that runs in polynomial time. We present, in the next section, an approximation algorithm for the densest-halfspace-covering problem which runs in polynomial time and does extremely well experimentally on both real and artificial data, although we are not yet able to prove its correctness under the uniform distribution.

4. Approximation algorithm for the densest-halfspace-covering problem

We have seen in the previous section that finding an approximation algorithm for the densest-halfspace-covering problem is of fundamental importance to PAC-learning halfspace intersections (and probably for other learning problems). Moreover, all constructive (or growth) neural net algorithms need such an approximation algorithm for training at the single-neuron level. Here we present our approach to this problem.

4.1. Description of the algorithm

We are, of course, concerned with the case where the data is not linearly separable because the linearly separable case is directly solvable by linear programming (LP). One way to approach this problem [12] is to try to minimize the perceptron criterion function [12] which can be converted to a linear cost function and, hence, solvable by LP. This, however, has nothing to do with minimizing the number of misclassifications, which is a truly nonlinear cost function of the weights. As a consequence, a set of weights that minimize the perceptron criterion function will, in general, contain too many misclassified examples, each being close to the separating hyperplane.

Another way to approach this problem is to use, incrementally, a LP procedure to try to incorporate one example at a time into a linearly separated data set. Hence, we will try the following greedy heuristic, which are call the incremental linear programming (IncLP) algorithm:

IncLP(S^+, S^-, L, H)
Parameters:
S^+: the set of positive examples.
S^-: the set of negative examples.
L: an initial set of negative examples separable from S^+ (may be empty).
H: an initial halfspace consistent with S^+ and L.
Output: (L, H) where L is subset of negative examples separable from S^+ and H is a halfspace consistent with S^+ and L.
Description: The algorithm builds on the set L by adding to it negative examples from S^-.

1. Set $R = S^-$.
2. If R is empty, Return (L, H).
3. Choose (possibly at random) an example x from R. Set $R = R - \{x\}$.
4. By using your favorite LP procedure, try to find a halfspace H^a consistent with $L \cup \{x\}$ and S^+.
5. If such halfspace exists Then set $L = L \cup \{x\}$ and $H = H^a$.
6. Go to step 2.

This procedure will return a halfspace H that covers a subset L of negative examples without covering any positive examples. It may happen, however, that a bad sequence of negative examples will be chosen such that the resulting subset L is small. Since the number of linearly separable dichotomies [10] increases exponentially with n, it is not feasible to find all of them. So we must instead look for ways to find, with high probability, large linearly separable subsets.

The first procedure that we might try (call it `mult_IncLP`), simply consists of running IncLP a certain number of times on the same training set. After each pass we record the halfspace found along with the size of the subset covered and we change randomly the order of the examples in the training set for the next pass. Then `mult_IncLP` returns the halfspace that covers the largest subset found from these multiple attempts. However, it

may happen that, with high probability, only small a subset will be found by IncLP at each attempt, causing mult_IncLP to return only a small subset. This means that each time a small subset is returned by IncLP, we must avoid loading this group of examples. Hence, a good 'rule of thumb' would be to remove these negative examples from a working set W and use IncLP to find another subset from the remaining examples in W. This is repeated until no negative example is left in W. At the end, we just retain the largest subset found. Hence, instead of mult_IncLP, we propose the following heuristic for our approximation algorithm:

Find_large_neg_subset(S^+, S^-)
Parameters:
S^+: the set of positive examples.
S^-: the set of negative examples.
Output: (L_m, H_m) where L_m is a (hopefully large) subset of negative examples linearly separable from S^+ and H_m is a halfspace consistent with L_m and S^+.
Description:
1. Set $W = S^-$ (W will hold the remaining negative examples).
   Set $U = \emptyset$ (U will hold the removed negative examples).
   Set $L_m = \emptyset$ (L_m will hold the largest subset found).
2. If W is empty, Return L_m.
3. Set $L = \emptyset$ and $H = +1$ (i.e. H is the constant halfspace).
4. (L, H) = IncLP(S^+, W, L, H).
5. Set $W = W - L$.
6. (L, H) = IncLP(S^+, U, L, H).
7. If $|L| > |L_m|$ Then $(L_m, H_m) = (L, H)$.
8. Set $U = U \cup L$. Go to step 2.

Note that, in addition to what has been said above, the algorithm tries (in step 6) to build a larger linearly separable set by trying to include the negative examples belonging to the subsets already found. More generally, it is clear that we can use IncLP in a similar way to find a large linearly separable subset of $S^+ \cup S^-$ with the constraint that a certain subset (not necessarily S^+) must be present in the solution. We will present, in section 6, numerical results that indicate the superiority of Find_large_neg_subset over mult_IncLP and the pocket algorithm.

Baum [3] has suggested that no 'good' incremental approximation algorithm exists for the densest-halfspace-covering problem. His arguments can be summarized as follows: for n large, with very high probability, any random set of n negative examples (on n inputs) will be separable from the positive examples. Because of that, it provides little information that one is getting a 'good' halfspace. So essentially, no further negative examples can be added to this set. Indeed, it is easy to imagine different scenarios where this approximation algorithm would give only a very small linearly separable subset. For example, suppose that the target function is the intersection of two halfspaces. Suppose also that the distribution of m^- negative examples is correlated to the distribution of positive examples as follows. Let Π be the polytope formed by the positive examples. For each face of Π, we have two (and only two) negative examples such that any hyperplane that separates *both* of them from Π, cannot separate any other negative example. Hence, if IncLP always starts with two such examples, the approximation algorithm will return subsets containing only two examples. This will cause the greedy heuristic to give $m^-/2$ halfspaces. However, all such catastrophic scenarios that we can think of have one thing in common that makes them very unlikely to occur in practice: the distribution of positive examples is *correlated* with the

distribution of negative examples which is also correlated with our (random) strategy for choosing the examples in the IncLP algorithm. That is, we have a malicious distribution [4] of examples controlled by adversaries. Hence, we think that our approximation algorithm will do well for simple function classes like halfspace intersections under the uniform distribution of examples and probably also under other reasonable [2] distributions. Our numerical simulations in section 6 suggest that this is indeed the case for both real and artificial data.

In the worst case, our approximation algorithm runs in polynomial time if we use Karmarkar's [26] polynomial time algorithm for LP. We have, however, used the Simplex algorithm [11] since, although its running time can increase exponentially with n in the worst case, it is reputed to be very efficient in practice. In fact, Smale [42] showed that the number of pivot operations needed to solve a LP problem is almost always no larger than the number of variables or the number of constraints, whichever is the larger. We have implemented the algorithm described in chapter 2 of [31], which uses Bland's rule to avoid cycling, and have encountered no problems.

Finally, there exists another approximation algorithm, known as the pocket algorithm [15], to find the largest separable subset of examples. One feature that might make this algorithm attractive is its 'convergence' theorem: *given a set of examples and a probability $p < 1$, there exists an N such that after $l \geqslant N$ iterations of the pocket algorithm, the probability of finding the largest separable subset of examples exceeds p.* This theorem, however, is not a strong statement since it gives no upper bound on the number N of iterations needed. Indeed, as Gallant himself emphasizes [15], the proof relies on the fact that there is a finite (but very small) probability that the examples from the largest (or a largest) linearly separable subset will be picked repeatedly by the perceptron which, following the perceptron convergence theorem, will find the corresponding optimal set of weights. Of course, we also have a similar 'convergence' theorem if we use our approximation algorithm repeatedly, each time changing (at random) the order of the examples. But in practice, we find it sufficient to use our approximation algorithm only once. An experimental comparison of the two approaches is given in section 6.1.

5. Learning neural decision lists

5.1. Binary classification problems

Here we extend the greedy method given in section 3 to the case of neural decision lists. The additional problem that arises now is that the set of positive examples is not necessarily convex (see figure 2). Hence, the greedy method will need to alternate between covering (cutting) positive subsets and negative subsets.

Consider the example of figure 2(a). Each time a halfspace covers a subset of examples, these are removed from the training set. For the first three halfspaces, only negative examples can be covered. Then, the positive examples will be covered by the next three halfspaces. Finally, the last halfspace will cover the remaining examples which are all of the same target: negative in this case.

We therefore propose the following greedy method for learning NDLs:

Build_NDL(S^+, S^-)

1. Let S^+ be the set of positive examples and S^- be the set of negative examples.
2. If $S^+ = \emptyset$, append the pair $(H_r = 1, -1)$ to the decision list and stop.
3. If $S^- = \emptyset$, append the pair $(H_r = 1, +1)$ to the decision list and stop.

4. Find the halfspace H^+ that covers the largest number of examples in S^+ and none of the examples in S^-. Let f^+ be the *fraction* of examples from S^+ covered by H^+.
5. Similarly, find the halfspace H^- that covers the largest number of examples in S^- and none of the examples in S^+. Let f^- be the *fraction* of examples from S^- covered by H^-.
6. If $f^+ > f^-$ Then append the pair $(H^+, +1)$ to the decision list and remove from S^+ the examples that are covered by H^+.
   Else append the pair $(H^-, -1)$ to the decision list and remove from S^- the examples that are covered by H^-.
7. Go to step 2.

Note that, at each step, we have decided to retain the halfspace that covers the largest *fraction* of the remaining positive (negative) examples. The reason is that the fraction of examples reflects more closely the probability measure we are covering than the actual number of examples.

5.2. Multi-class problems

There are several ways one can extend this approach to multiple-class functions. Say that we have Q classes so that the function to learn $\tau(x)$ can take any value from $1, \ldots Q$. One way to build the multiple class NDL is to find, at each step, the halfspace that covers (cuts) the largest fraction of examples *from one class only*. The following heuristic builds a NDL for multi-class problems:

1. For $\tau = 1, \ldots Q$, let S^τ be the set of examples of the class τ.
2. If *all* but one of the S^τ are empty, append the pair $(H_r = 1, \tau_0)$ to the decision list and stop (τ_0 is the class of the non-empty set).
3. For $\tau = 1, \ldots Q$:
   If $S^\tau \neq \emptyset$, find the halfspace H^τ that covers the largest fraction f^τ of S^τ and none of the examples in $\cup_{\sigma \neq \tau} S^\sigma$.
4. Choose the H^σ that covers the largest fraction f^σ and remove these examples from S^σ. Append to the decision list the pair (H^σ, σ).
5. Go to step 2.

We can think of other, and probably better, ways to use these halfspace-covering heuristics in multi-class situations by incorporating some knowledge about the dispersion of the data. We could, for example, perform $\lceil \log_2(Q) \rceil$ different dichotomies of the Q classes and then use, in parallel, the Build_NDL algorithm of the last section to create $\lceil \log_2(Q) \rceil$ different binary classification NDLs. These dichotomies could be done by performing a principal component analysis on the data or by using any other criteria like those presented in [41]. The point we want to make is that our approach is flexible enough to be applied in a rich variety of ways to multi-class problems.

6. Experimental results

6.1. The approximation algorithm versus the pocket algorithm

Because most of the constructive algorithms use the pocket algorithm at the single-neuron level to find the 'optimal' halfspace, we include here a numerical comparison of our approximation algorithm with the pocket algorithm on both linearly and nonlinearly separable functions.

The linearly separable case. In this case, the approximation algorithm requires only one pass of IncLP. We took great care in implementing the pocket as explained in the references cited above.

For each test, we generate at random a hyperplane in the $[-1,+1]^n$ region. The training examples were drawn randomly in $[-1,+1]^n$ and classified according to this target function. Because both algorithms are guaranteed to converge to a solution, the performance criteria will be the time needed to find it. The average CPU time taken on a 1.4 MFLOPS computer (YARC's NuSuper accelerator board for the Macintosh) is reported in figure 3. Although the CPU time depends on both the machine and the code written, it is a good measure of the *relative* efficiency of the algorithms. One can see that the approximation algorithm outperforms the pocket by many orders of magnitude. Whereas the approximation algorithm scales linearly with the number of examples in the training set, the pocket clearly scales super-linearly with the number of examples. This can be explained by the fact that because the pocket chooses examples at random, it will spend most of the time checking examples already well classified.

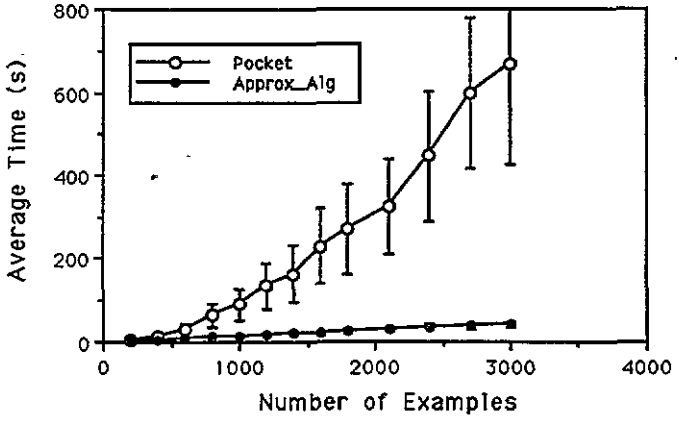

Figure 3. Comparison of the approximation algorithm with the pocket algorithm on linearly separable functions for $n = 10$. Each point on the graph is the average over 100 tests. Also shown are the standard deviations.

The nonlinearly separable case. One of the simplest nonlinearly separable functions is the intersection of two parallel halfspaces. So, we tested both algorithms on this problem. The task here is to classify the largest possible fraction of the negative examples, keeping all the positive examples well classified. Each point on the graph of figure 4 is the average number of negative examples in the largest linearly separable subset found by the algorithm. The average is done over five different target networks; each being tested ten times (a total of 50 tests for each point of the graph). The input (instance) space is the 16-dimensional Euclidean region $[-1,+1]^{16}$. Each example of the training set is generated uniformly in this input space and classified according to the target function. Each target network consists of an intersection of two parallel halfspaces, a distance of 0.5 apart, randomly oriented and centred at the origin of the input space. Hence the optimal plane for this target function covers half of the total negative measure.

We have compared three algorithms: (i) our approximation algorithm `Find_large_neg_subset`; (ii) the pocket algorithm with rules (each positive example is taken as a rule that must not be violated); and (iii) `mult_IncLP` described in section 4.1. Here the relevant

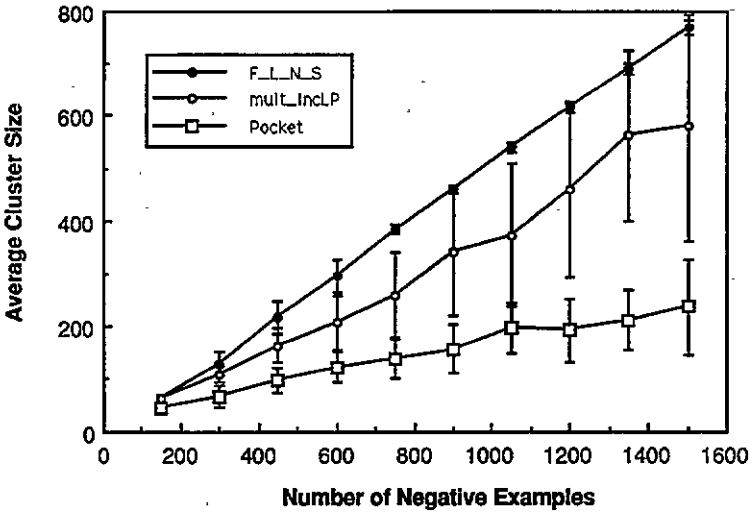

Figure 4. Comparison of the approximation algorithm `Find_large_neg_subset` with the pocket algorithm and with `mult_IncLP` (see text) on nonlinearly separable functions for $n = 16$. The average 'cluster size' is the average number of negative examples in the largest linearly separable subset found. Each point on the graph is the average over 50 tests. The error bars indicate the standard deviations.

violated); and (iii) `mult_IncLP` described in section 4.1. Here the relevant question is: given the same amount of time and the same training set, which algorithm returns the largest cluster (subset) of negative examples? To answer this, we first run `Find_large_neg_subset` on a training set to find a subset (cluster) of negative examples and note both the time spent by the algorithm and the size of the cluster returned. Then, we run the pocket with rules for the same amount of time and on the same training set and we note the size of the cluster returned. Finally we run `mult_IncLP` for the same amount of time taken by `Find_large_neg_subset` and record the size of the largest cluster found (among the multiple passes of `IncLP`).

The results are plotted in figure 4 for the three algorithms. Clearly, `Find_large_neg_subset` outperforms the pocket algorithm. The fact that it also outperforms `mult_IncLP` provides strong numerical evidence for the importance of removing bad sequences of negative examples when they are found. This is the basic difference between `Find_large_neg_subset` and `mult_IncLP`. Note also the large error bars (standard deviation of the cluster size returned) of `mult_IncLP` and the very small ones for `Find_large_neg_subset`. This shows that `mult_IncLP` is much less reliable and consistent than `Find_large_neg_subset`. Also, we must mention that, for a training set of 300 or more negative examples, `Find_large_neg_subset` returned, on average, an optimal cluster containing half of the total number of negative examples in the training set.

Hence, from these results, we conclude that `Find_large_neg_subset` is superior, on average (and almost always), to `mult_IncLP`.

6.2. Learning halfspace intersections

In this section, we describe the results of the extensive experimental tests performed on random halfspace intersections.

Our class of functions to learn will be the class of h halfspace intersections or

equivalently, the FFN containing one layer of h hidden units and one output unit implementing an 'AND' of the hidden layer. We want to find out how the performance of the algorithm scales with the number of examples in the training set, the dimension of the input space and the number of halfspaces in the target net function.

For each test, the target net function is generated as follows: the value of each connection going from an input unit to a hidden unit is chosen uniformly and randomly in the interval $[-1, +1]$. The bias of the hidden units are also chosen randomly in the interval $[-1, +1]$. The output unit is hardwired to compute an AND of its inputs. We make sure that the region defined by the halfspace intersection is not void. This gives our target net function. The training examples were drawn randomly in $[-1, +1]^n$ and classified according to the target function. The average number of examples in the training set varies approximately between 100 and 2000 points. Another separate set of approximately 2000 examples is drawn according to the same distribution, classified according to the target function, and used as a test set for the generalization ability of our algorithm.

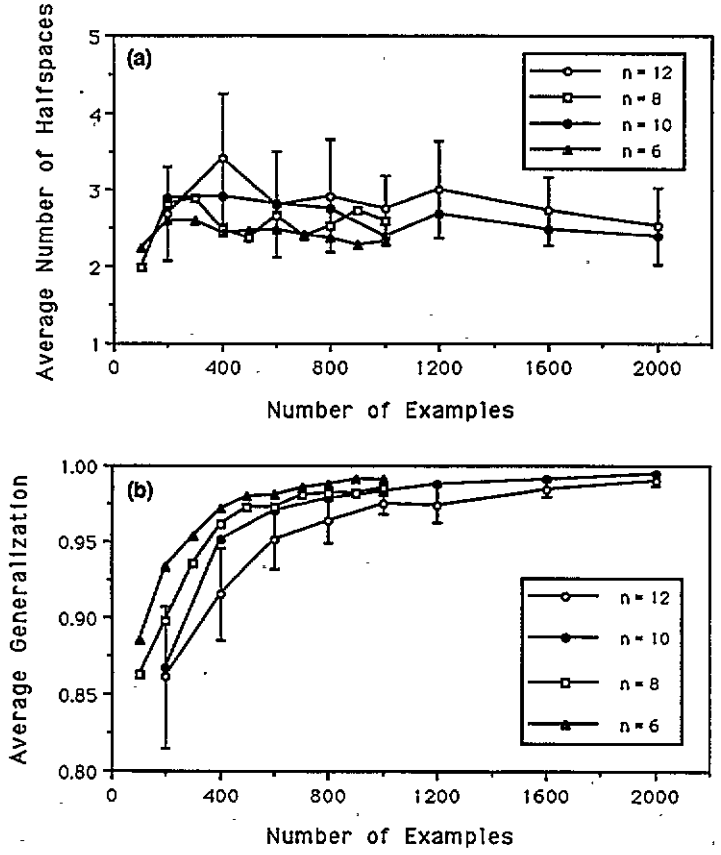

Figure 5. (*a*) The average number of halfspaces and (*b*) the average generalization of the hypothesis net when learning intersections of two halfspaces. For clarity, only standard deviations for $n = 12$ are shown.

We tried target nets with $n = 6, 8, 10, 12$ and $h = 2, 4, 6$. For each pair of values (n, h), the number of examples in the training set was varied between approximately 100 and 2000. The number of examples in the test set was kept constant at around 2000 examples. Typical

results are shown in figures 5 and 6. Each point on these figures represents the average over 100 tests. We present both the average number of halfspaces in the hypothesis net returned by the algorithm and its generalization ability for different values of (n, h). From figure 5 and figure 6, one can see that, on average, the algorithm is doing well. Not only is the number of halfspaces returned by the algorithm small enough to allow good generalization, but it also does scale very nicely with the size of the problem.

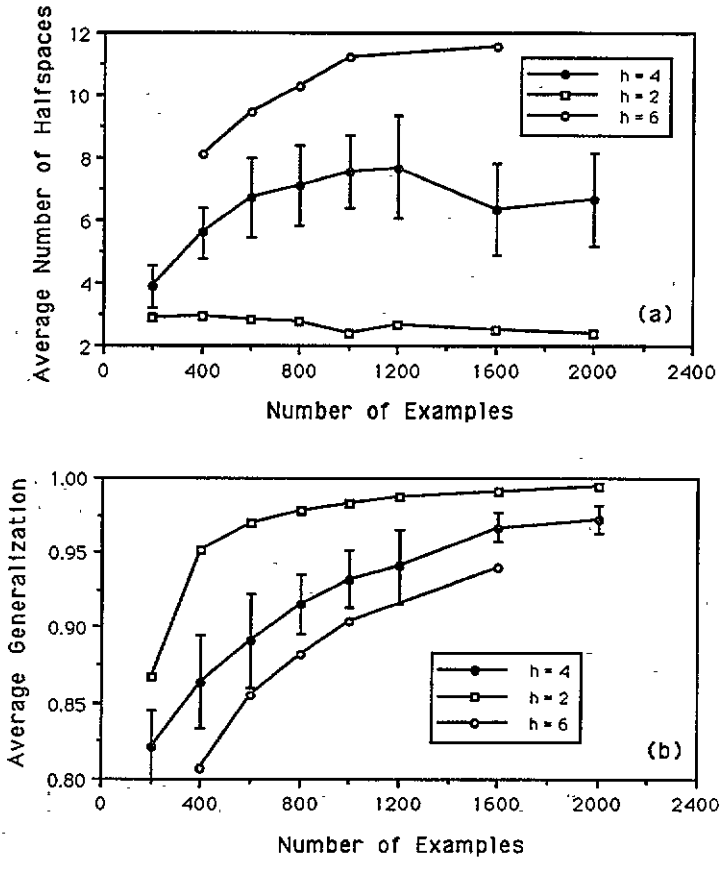

Figure 6. (*a*) The average number of halfspaces and (*b*) the average generalization of the hypothesis net when learning intersections of $h = 2, 4, 6$ halfspaces for $n = 10$. For clarity, only standard deviations for $h = 4$ are shown.

We have also investigated the worst case behaviour of the algorithm over the set of tests performed. For each pair (n, h), we looked at the *maximum* number of halfspaces returned by the algorithm at any test. This number was 4 for (8, 2), 5 for (10, 2) and 10 for (10, 4). This indicates that cases where the algorithm will perform poorly will be encountered very rarely. It would be interesting to investigate this question theoretically.

To avoid the objection that these are moderate-sized problems, we run tests on (20, 2), (30, 2), (35, 2), (40, 2), and (50, 2). The results are presented in table 1. These tests were done as follows. We first generate at random a halfspace intersection function made of two n-dimensional halfspaces, orthogonal to each other and passing through the origin† of

† Of course, this information is not coded in the learning algorithm.

Table 1. The number of examples (m) and the number of halfspaces (k) needed by the greedy heuristic to get a generalization rate (Gen) as a function of the dimension (n) of the input space for an intersection of two halfspaces. M is the size of the test set. These results are averages over N runs. The best generalization rate (Opt-gen) was obtained by executing the hypothesis HI on r ($\leqslant k$) nodes. The approximate CPU time is that obtained from an IBM RS/6000 machine.

n	m	k	Gen	r	Opt-gen	M	N	CPU time
10	3000	4.3	99.4% ± 0.1%	3.6	99.5 ± 0.2%	10000	10	~ 0.3 h
20	4000	5.0	98.1% ± 1.0%	3.3	98.8 ± 0.3	20000	6	~ 0.7 h
30	6000	4.3	98.6% ± 0.1%	3.6	98.9 ± 0.1%	20000	5	~ 1.6 h
40	10000	5	98.2%	4	98.6%	40000	1	~ 12 h
50	14000	5	98.8%	4	99.0%	40000	1	~ 36 h

the continuous input space $[-1,+1]^n$. Then a training sample consisting of $m/2$ positive examples and $m/2$ negative examples is generated uniformly at random (note that the positive measure is $\frac{1}{4}$ of the total measure for these target functions). When the greedy algorithm returns a hypothesis function with k halfspaces, the generalization is tested separately on $M/2$ positive and $M/2$ negative new examples generated uniformly. The average (Gen) of these two scores was reported. Since the last halfspace of the hypothesis often cuts only a few points, we have also reported the number of nodes r for which the hypothesis exhibits the best generalization (Opt-gen).

Table 2. The mirror symmetry problem in 30 dimensions. The training set consisted of m examples (half of them were positive) generated uniformly on the hypercube. The generalization rate (Gen) was tested on 4000 examples (half of them positive) generated uniformly. Each value represents an average over 20 different instances and σ is the standard deviation of the generalization rate.

m	100	200	400	600
⟨Gen⟩	69.7%	80.1%	90.8%	94.4%
σ	7.5%	3.5%	1.7%	0.9%

The number of examples and halfspaces needed to get a generalization rate of 98–99% is consistent with a polynomial increase with respect to the dimension of the input space (for $n \leqslant 50$). These results do not show any evidence of an exponential increase in the number of examples needed to learn halfspace intersections (as conjectured by Baum [3]). Hence we think that these experimental results strongly suggest that the greedy method learns halfspace intersections under the uniform distribution.

Table 3. The generalization rate (Gen) of our greedy method, standard deviation (±) and the number of nodes found (k) for various data sets. n is the number of inputs (dimension) and m, the number of examples. Each result is averaged over 20 trials. The training set consists of the $\frac{2}{3}$ of the m examples and the test set consists of the $\frac{1}{3}$ remaining. We also include the results of C4. n is the number of input variables.

Dataset	m	n	Def-acc	C4	⟨Gen⟩	⟨k⟩
CH	3196	36	52.2%	99.2% ± 0.3	95.2% ± 0.5%	4.0
G2	163	9	53.4%	74.3% ± 6.6	76.4% ± 6.7%	4.7
IR	150	4	33.3%	93.8% ± 3.0	95.1% ± 6.3%	3.4
V0	435	16	61.4%	95.6% ± 1.3	92.0% ± 2.8%	2.0
V1	435	15	61.4%	89.4% ± 2.5	87.3% ± 2.3%	3.4

An interesting function that has been the subject of controversy [19, 39] is the *mirror symmetry detector*. Here the target output is $+1$ if and only if the second half of the input bits is a mirror reflection of the first half [19]. This function is known to have at least two different neural net representations: an intersection of two parallel halfspaces and a majority function of n order-two-disjuncts. Since the latter is not a NDL, the greedy always found a halfspace intersection. Again the training was done on $m/2$ positive and $m/2$ negative examples obtained from the uniform distribution on $\{-1, +1\}^n$. Testing was done on both $M/2$ positive and $M/2$ negative examples obtained from the same distribution. As the results show (see table 2), the greedy method was able to find a good approximation of this function after seeing very few examples. The function returned consisted always of two, almost parallel, halfspaces. For the sceptic, we have done five runs in 50 dimensions and have obtained an average generalization rate of 97.5% after training on only $m = 1600$ examples! Testing was done on $M = 20\,000$ examples obtained from the same distribution. Note that it is important here to test separately on both the positive and the negative measure. Otherwise, a trivial net giving -1 to all examples would achieve almost perfect generalization on the uniform distribution since the positive region has only a fraction of $2^{-n/2}$ of the total measure. But here this trivial net gets only 50% generalization on the testing set. These results show how easy learning can be for a reasonable distribution: here the returned net only needs to be correct on inputs having roughly an equal amount of $+1$ and -1 entries. For this reason, the returned weight values do not increase exponentially.

To summarize, our numerical results strongly suggest that the algorithm is behaving as 'Occam's algorithm' and does learn halfspace intersections under the uniform distribution.

6.3. Real data sets

We have tested our greedy heuristic of section 5 to build NDLs on data sets taken from a collection distributed by the machine learning group of the University of California at Irvine†. These sets were kindly provided to us by Robert Holte who studied recently [22] the performance of certain machine learning algorithms on these data sets. The results of our algorithm are summarized in table 3. We have included, for comparison, the performance of C4 on these data sets as reported recently by Holte [22]. C4 is a 'state of the art' tree-induction algorithm [34] capable of producing very complex decision rules. Also indicated is the 'default accuracy' one has when classifying all the testing examples according to the class containing the largest number of examples. We now comment on our results:

CH: *Chess end-game*. This data set was originally generated and described by Shapiro [40]. The goal is to determine whether or not a given chess board configuration is a winning position for the white player. The white player has a king and rook whereas the black player has a king and pawn where the pawn is located on 'a7' (just ready to be promoted to a Queen). The input consists of 35 binary features and one ternary feature describing a board position of a chess end-game. The output is whether or not the position is a winning one for the white player. These features, as described in Shapiro's book [40], have been explicitly engineered for C4 by a chess expert working with a version of C4 built specially for this purpose. In view of the importance of representations, it is surprising to see that our algorithm achieves a 95% generalization score on 'C4–hand-crafted data'.

G2: *Glass identification*. Here the goal is to determine if a given piece of glass is 'float-processed' or 'non-float-processed'. The study of classification of types of glass was motivated by criminological investigation. At the scene of the crime, the glass left can be

† Contact person: Pat Murphy (pmurphy@ics.uci.edu).

used as evidence... if it is correctly identified! The input vector consists of 9 continuous-valued attributes where each attribute indicates the concentration of a given element (Mg, Na, Al ...) and one attribute indicates the refractive index. The performance of our greedy method was only slightly (but not significantly) better than C4. Although we think that this kind of problem is well suited for our approach (all attributes are continuous), the number of examples is not enough to achieve a higher accuracy.

IR *Iris data set.* This is perhaps the best known database to be found in the pattern recognition literature [12]. The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant: virginica, versicolor or setosa. One class (setosa) is linearly separable from the other two; the latter are not linearly separable from each other. The four inputs are continuous-valued and correspond to the length and width of the sepal and petal. Our algorithm achieved good accuracy on this set and slightly (but not significantly) better than C4.

V0–V1: *United States congressional voting records database.* The V0 data set includes votes for each of the US House of Representatives Congressmen on 16 key votes (Salvador aid, aid to Nicaraguan Contras, education spending, MX-missiles, ...). The result of each vote is expressed as a ternary attribute: yea, nay or '?' where this last value is taken when a congressmen did not vote, voted present or voted present to avoid conflict of interest. This is a two-class problem since a Congressman is either Democrat or Republican. The V1 data set is identical to V0 except that the most informative attribute (physician fee freeze) has been deleted, which makes the problem harder. The performance of our algorithm for these sets is slightly less than that of C4, but not significantly.

Hence, the greedy method for constructing NDLs can handle with success these real data sets with roughly the same level of performance as C4. We must mention, however, that the version of C4 used in these tests incorporates pruning. We could probably also improve our results by pruning the NDLs and by using a cross-validation data set to test each new halfspace found during the building process. Moreover, once a halfspace is found, we could 'fine-tune' it with a quadratic programming algorithm so as to find a hyperplane with maximal stability. In short, this approach is flexible enough to incorporate many engineering tricks that have proven useful in the past. We are investigating these possibilities.

7. Summary

By formulating the problem of halfspace intersections as a set-covering problem, we were able to bring it down to the much simpler, yet very difficult, sub-problem: given a set of nonlinearly separable examples, find the largest linearly separable subset of it. Short of solving the CAP, the only hope for learning halfspace intersections is to find a good approximation algorithm for this NP-hard subproblem.

The approximation algorithm given in this paper seems to work well in practice. At the single-neuron level, it outperforms the pocket algorithm used by many constructive algorithms on both linearly and nonlinearly separable functions.

The greedy method which incorporates this approximation algorithm seems able to learn halfspace intersections under the uniform distribution of examples. Whereas one can always think of malicious situations where the approximation algorithm will do poorly, the numerical results are very encouraging and indicate that this will occur very rarely in practice. The problem of whether or not there exists an approximation algorithm, with a performance guarantee in the worst case, is still open. We think that this question is

of fundamental importance because any algorithm that can give a number of halfspaces bounded by $h^{\beta}m^{\alpha}$, for $\alpha < 1$ and $\beta \geqslant 1$, will satisfy the Occam's razor criteria.

We extended the approach to a larger and richer class of concepts, namely the class of neural decision lists. Again the approach seems to work well. Our results on real-data problems suggest that concepts like halfspace intersections (unions) and NDLs are not only important from the theoretical point of view, but also useful in practice.

Acknowledgments

We are very grateful to our colleague Gary Slater, to IBM system engineering representative Toby Taylor and to IBM for giving us precious CPU time on their IBM RS/6000 workstations. Also we thank Robert Holte for providing us the data sets to test our greedy heuristics. One of us (MM) would like to thank Pál Ruján and Eric Baum for several stimulating discussions. This work has been supported by NSERC grant OCG-0 042 038.

Appendix

In this section we show how to construct the equivalent FFN of figure 2 from the NDL obtained from our algorithm described in section 5.

For such a FFN, if we denote by $v_{j,k}$ the value of the weight-connecting hidden unit k to hidden unit j, the output S_j^{μ} of hidden unit j, upon presentation of pattern $\boldsymbol{x}^{\mu}$ ($\mu = 1, \ldots m$), is given by

$$S_j^{\mu} = \operatorname{sgn}\left(\sum_{i=0}^{n} w_{j,i} x_i^{\mu} + \sum_{k=0}^{j-1} v_{j,k} S_k^{\mu}\right) \tag{A1}$$

where $\operatorname{sgn}(x) = +1$ if $x > 0$, and -1 otherwise. The cascade architecture is reflected, in this equation, by the fact that hidden unit j is updated only after all hidden units $k < j$ have been updated.

The $w_{j,i}$s are readily obtained from the halfspaces of the NDL. What remain to be found are the connections between the hidden units. Recall from section 2 that our goal is to try to get a set of internal representations of the type $-1, -1, \ldots, -1, +1, -1, \ldots, -1$.

To assure that hidden unit j inhibits all hidden units $i > j$ when $S_j^{\mu} = +1$, we choose to connect unit j to all units $i > j$ in the following way:

$$v_{i,j} = -\sum_{k=0}^{n} |w_{i,k}| - 1 \qquad \text{for} \quad j = 1, \ldots h \quad \text{and} \quad i = j + 1, \ldots h \tag{A2}$$

where h denotes the number of hidden units or, equivalently, the number of nodes of the NDL. However, the drawback of the above choice is that unit j will force all the other units $i > j$ to output $+1$ whenever $S_j^{\mu} = -1$ whereas we want each unit i to output $+1$ only when *all* other unit $j < i$ output -1 *and* when $\boldsymbol{x}^{\mu}$ is in its positive halfspace. This can be corrected by renormalizing the biases of each hidden unit according to

$$w_{i,0} \to w_{i,0} + \sum_{j=1}^{i-1} v_{i,j} \qquad \text{for} \quad i = 1, \ldots h \tag{A3}$$

without affecting the inhibitory action that unit j has on all the units above it when $S_j^\mu = +1$. Hence, we have achieved the desired set of internal representations.

Finally, we have to set the value of each weight u_j connecting hidden unit j to the output unit such that its output always gives the desired value. If we denote by τ_j the target of the patterns covered by the positive halfspace of hidden unit j, we can achieve this goal by the following choice:

$$u_j = \tau_j \qquad \text{for} \quad j = 1, \ldots h \tag{A4}$$

for the connections and setting the bias according to

$$u_0 = \sum_{j=1}^{h} \tau_j \,. \tag{A5}$$

Because of the structure of the set of internal representations, the output will be $\mathrm{sgn}(2 \times \tau_i)$ whenever a pattern with target τ_i is presented to the net. Hence, all the examples in the training set will always be correctly classified by this cascade FFN (equivalent to the NDL).

For multiple-class problems, we will again choose the inter-hidden units connections and bias according to (A1) and (A2) so as to have the same set of internal representations as previously. But now each hidden unit is being associated with a class value. Moreover, since the output units are perceptrons, we need to represent each class τ by a binary vector $\boldsymbol{\tau}$. We have, of course, complete freedom to choose any binary representation we want. If we now denote by ν the class index ($\nu = 1, \ldots, Q$) and by $\boldsymbol{\tau}^\nu$ its binary representation, the weight u_i^ν connecting any hidden unit that covers patterns of the class ν to output unit i is given by

$$u_{i,\nu} = \tau_i^\nu \tag{A6}$$

and the bias for output unit i now generalizes to

$$u_{i,0} = \sum_{j=1}^{h} \tau_i^j \tag{A7}$$

where $\boldsymbol{\tau}^j$ denotes the vector representation of the examples covered by hidden unit j. Again, we have the guarantee that each example in the data set will be correctly classified.

References

[1] Angluin D 1988 Queries and concept learning *Machine Learning* **2** 319

[2] Bartlett P L and Williamson R C 1991 *Proc. Fourth Workshop on Computational Learning Theory* (Palo Alto, CA: Morgan Kaufmann) p 24

[3] Baum E B 1990 On learning a union of halfspace *J. Complexity* **6** 67

[4] Baum E B 1990 The perceptron algorithm if fast for nonmalicious distributions *Neural Computation* **2** 248

[5] Baum E B 1990 A polynomial time algorithm that learns two hidden unit nets *Neural Computation* **2** 510

[6] Blum A and Rivest R L 1988 Training a 3-node neural network is NP-complete *Proc. First Workshop on Computational Learning Theory* (Palo Alto, CA: Morgan Kaufmann) p 9

[7] Blumer A, Ehrenfeucht A, Haussler D and Warmuth K 1989 Learnability and the Vapnik–Chervonenkis dimension *J. ACM* **36** 929

[8] Blumer A, Ehrenfeucht A, Haussler D and Warmuth K 1987 Occam's razor *Information Proc. Lett.* **24** 377

[9] Chvatal V 1979 A greedy heuristic for the set-covering problem *Math. Oper. Res.* **4** 3

[10] Cover T M 1965 Goemetrical and statistical properties of systems of linear inequalities with applications in pattern recognition *IEEE Trans. Electron. Comput.* **EC-14** 326
[11] Dantzig G B 1963 Linear programming and extensions (Princeton, NJ: Princeton University Press)
[12] Duda R O and Hart P E 1973 *Pattern Classification and Scene Analysis* (New York: Wiley)
[13] Fahlman S E and Lebière C 1990 The cascade-correlation learning architecture *Adv. Neural Information Processing Systems* **2** 524
[14] Frean M 1990 The upstart algorithm: A method for constructing and training neural networks *Neural Computation* **2** 198
[15] Gallant S I 1990 Perceptron-based learning algorithms *IEEE Trans. Neural Networks* **NN-1** 179
[16] Gary M R and Johnson D S 1979 *Computers and Intractability, A Guide to the Theory of NP-completeness* (New York: Freeman)
[17] Golea M, Marchand M, and Hancock T 1992 On learning μ-perceptron networks with binary weights *NIPS*92* to appear
[18] Golea M and Marchand M 1990 A growth algorithm for neural network decision trees *Europhys. Lett.* **12** 205
[19] Minsky M and Papert S 1988 *Perceptrons* (Cambridge, MA: MIT Press)
[20] Hancock T, Golea M, and Marchand M 1991 Learning nonoverlapping perceptron networks from examples and membership queries *Harvard University Preprint* TR-26-91 (Submitted to *Machine Learning*)
[21] Haussler D 1988 Quantifying inductive bias: AI learning algorithms and Valiant's learning framework *Artif. Intell.* **36** 177
[22] Holte R C 1991 Very simple classification rules perform well on most data sets *TR-91-16* Computer Science, University of Ottawa (to appear in *Machine Learning*)
[23] Johnson D S 1974 Approximation algorithms for combinatorial problems *J. Comput. System Sci.* **9** 256
[24] Johnson D S and Preparata F P 1978 The densest hemisphere problem *Theor. Comput. Sci.* **6** 93
[25] Judd J S 1990 *Neural Network Design and the Complexity of Learning* (Cambridge, MA: MIT Press)
[26] Karmarkar N 1984 A new polynomial time algorithm for linear programming *Combinatorica* **4** 373
[27] Lin J H and Vitter J S 1991 Complexity results on learning by neural nets *Machine Learning* **6** 211
[28] Marchand M, Golea M, and Ruján P 1990 A convergence theorem for sequential learning in two-layer perceptrons *Europhys. Lett.* **11** 487
[29] Mézard M and Nadal J-P 1989 Learning in feedforward neural network: the tiling algorithm *J. Phys. A: Math. Gen.* **22** 2191
[30] Nadal J-P 1989 Study of a growth algorithm for a feedforward network *Int. J. Neural Systems* **1** 55
[31] Papadimitriou C H and Steiglitz K 1982 *Combinatorial Optimization* (Englewood Cliffs, NJ: Prentice-Hall)
[32] Pitt L and Valiant L G 1988 Computational limitations on learning from examples *J ACM* **35** 965
[33] Preparata F P and Shamos M I 1985 *Computational Geometry* (New York: Springer)
[34] Quinlan J R 1986 Induction of decision trees *Machine Learning* **1** 81
[35] Rivest R L 1987 Learning decision lists *Machine Learning* **2** 229
[36] Rujan P 1992 A fast method for calculating the perceptron with maximal stability *Preprint* (submitted to *Neural Computation*)
[37] Ruján P and Marchand M 1989 Learning by minimizing resources in neural networks *Complex Systems* **3** 229
[38] Rumelhart D E, Hinton G E and Williams R J 1986 Learning representations by back-propagating errors *Nature* **323** 533
[39] Rumelhart D E and McClelland J L 1986 *Parallel Distributed Processing* vol 1 (Cambridge, MA: MIT Press)
[40] Shapiro A D 1987 *Structured Induction and Expert Systems* (Wokingham, UK: Addison-Wesley)
[41] Sirat J A and Nadal J P 1990 Neural trees: a new efficient tool for classification *Network* **1** 423
[42] Smale S 1983 On the average number of steps of the simplex method of linear programming *Math. Programm.* **27** 241
[43] Utgoff P E 1989 Perceptron trees: a case study in hybrid concept representation *Connection Sci.* **1** 377
[44] Valiant L G 1984 A theory of the learnable *Commun. ACM* **27** 1134
[45] Valiant L G and Warmuth M K 1989 The border augmented symmetric differences of halfspaces is learnable *unpublished*