From: World Congress on Neural Networks: Proceedings of the 1993 Annual Meeting of the International Neural Network Society, vol. 3, pp. 556–559, Hillsdale, NJ: Erlbaum Associates (1993).

An Approximation Algorithm to Find the Largest Linearly Separable Subset of Training Examples

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

Abstract

We present an approximation algorithm for the NP-hard problem of finding the largest linearly separable subset of examples among a training set. The algorithm uses, incrementally, a Linear Programming procedure. We present numerical evidence of its superiority over the Pocket algorithm used by many neural net constructive algorithms.

1 Introduction

In this paper, we present an approximation algorithm for finding the largest linearly separable subset of examples among a training set. The study of this problem is motivated by the recent interest in *constructive algorithms* [8, 4, 3, 2]. These are algorithms that construct a feedforward network from examples by training individual perceptrons on some carefully chosen subset of examples. Hence the success of these algorithms strongly depends on the performance of the single perceptron training procedure they use. Moreover, it has been shown recently [7] that if one can always find the largest linearly separable subset of examples in a training set, then there exists some constructive algorithms that can provably learn the class of functions representable as *Neural Decision Lists* [7]. Unfortunately, finding the largest linearly separable subset of examples among a training set is NP-hard since it contains, as a particular case, a NP-complete problem known as the *Densest Hemisphere* [5]. Since we cannot solve this problem in the worst case, we present an approximation algorithm that seems to give good results in practice under reasonable distributions of examples like the uniform one. Because it is a Linear Programming (LP) problem to find if a data set is linearly separable, and since LP problems can be solved efficiently, our approximation algorithm uses, incrementally, a LP algorithm. We present numerical evidence for the superiority of this method over the Pocket algorithm [3], used by many constructive neural net algorithms.

2 The Approximation Algorithm

Let us first define our notation. Each input example is a n-dimensional vector with real valued components. A *linear threshold function* is specified by n real valued weights w_i and a single real valued bias w_0. The output of this function is $+1$ or -1 depending on whether or not: $\sum_{i=1}^{n} w_i x_i + w_0 > 0$. Such functions are also referred to as *perceptrons* or *halfspaces*. We denote by H the positive halfspace $\{\vec{x} : \vec{w} \cdot \vec{x} + w_0 > 0\}$ and by $\overline{H}$ its complement. Halfspace H is said to *cover* example $\vec{x}$ if $\vec{x} \in H$. 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 $\overline{H}$.

We are of course concerned with the case where the data is not linearly separable, because the linearly separable case is directly solvable by LP. One way to approach this problem is to try to minimize the

perceptron criterion function [1] 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 propose the following greedy heuristic; call it the Incremental Linear Programming (IncLP) algorithm:

IncLP(L, H, S)
Parameters
L: an initial set of linearly separated examples (may be empty).
H: an initial halfspace consistent with L.
S: A set of training examples.
Output: (L, H) where H is a halfspace consistent with L.
Description: The algorithm builds on the set L by adding to it examples from S.

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

This procedure will return a halfspace H that is consistent with a subset L of training examples. It may happen, however, that a bad sequence of examples will be chosen such that the resulting subset L is small. Since the number of linearly separable dichotomies 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 (with random reordering of the examples) and returning the best solution found among the multiple attempts. However, it may happen that only small a subset will be found by **IncLP** at each attempt; causing **mult_IncLP** to return only a small subset. Each time that a small subset is returned by **IncLP**, this means that we must avoid loading this group of examples. Hence, a good "rule of thumb" would be to remove these examples from a working set W and use **IncLP** to find another subset from the remaining examples in W. This is repeated until no example is left in W. At the end, we just retain the largest subset found. Hence, instead of **mult_IncLP**, we propose the following approximation algorithm:

Find_large_subset(S, S^*, H^*)
Parameters
S: A set of training examples.
S^*: A set (may be empty) of linearly separable examples which must be included in the returned set L_m
H^*: A halfspace consistent with S^*
Output: (L_m, H_m) where L_m is a (hopefully large) subset of $S \cup S^*$ consistent with H_m.
Description:

1. Set $W = S$ (W will hold the remaining training examples).
   Set $U = \emptyset$ (U will hold the removed examples).
   Set $L_m = S^*$ (L_m will hold the largest subset found).
2. If W is empty, Return L_m.
3. Set $L = S^*$ and $H = H^*$

4. $(L, H) =$ **IncLP**$(L,\ H,\ W)$.
5. Set $W = W - L$.
6. $(L, H) =$ **IncLP**$(L,\ H,\ U)$.
7. **If** $|L| > |L_m|$ **Then** $(L_m, H_m) = (L, H)$.
8. Set $U = U \cup L$. Goto step 2.

Note that we have included an extra parameter S^*, in case we require that the solution contains a set S^* of examples (considered as *rules* not to be violated) that we want to be present in the solution L_m. Note also 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 examples belonging to the subsets already found. In the worst case, our approximation algorithm runs in polynomial time if we use Karmarkar's[6] polynomial time algorithm for LP. We have, however, used the Simplex algorithm which is reputed to be very efficient in practice. Finally, note that we can use **Find_large_subset** repeatedly on the same training set, with a random reordering of the examples, to try to find a better solution. This leads to a convergence theorem similar to that of the Pocket algorithm. In practice, we find it sufficient to use the approximation algorithm only once.

3 Numerical Results

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 non linearly separable functions.

The Linearly Separable Case. In this case, the approximation algorithm requires only one pass of IncLP. We took a great care in implementing the pocket as explained in [3]. 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, is reported in fig. 1(a). 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 choose examples at random, it will spend most of the time checking examples already well classified.

The Non Linearly Separable Case. Since one of the simplest non linearly separable function is the intersection of two parallel halfspaces, we tested our algorithm on this problem. Each target function consist of an intersection of two halfspaces defined by two parallel hyperplanes, a distance of 0.5 apart, randomly oriented and centered at the origin. Hence the optimal plane for this target function covers half of the total negative measure and all of the positive measure. Each training example is a n-dimensional vector generated uniformly in the Euclidean region $[-1, +1]^n$ and classified according to the target function. 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 that can be linearly separated from all the positive examples? We present the results in fig. 1(b) for three algorithms: i) our approximation algorithm **Find_large_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 above. For each test, we first run **Find_large_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 do the same for **mult_IncLP**. Each point on the graph of fig. 1(b) 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

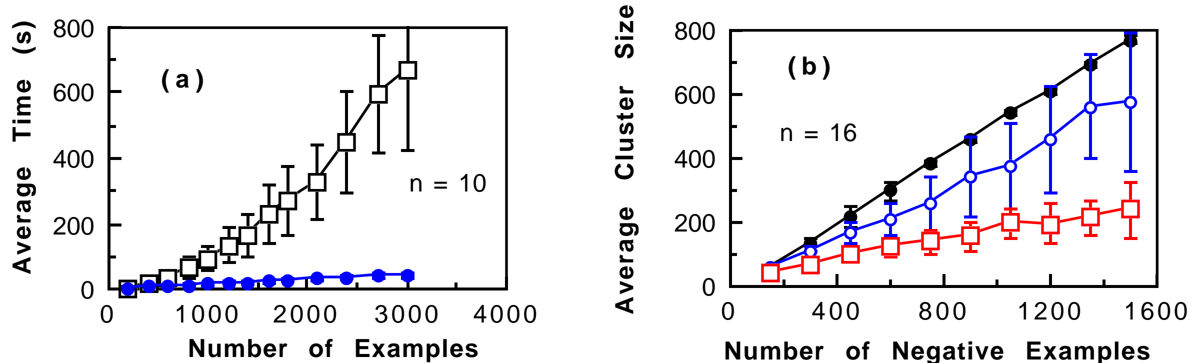

Figure 1: Comparison of the approximation algorithm **Find_large_subset** (full circles) with the pocket algorithm (squares) on (a) linearly separable functions for $n = 10$. Each point on the graph is the average over 100 tests. (b) On non linearly separable functions for $n = 16$. The average "cluster 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. The open circles are for **mult_IncLP**.

ten times (a total of 50 tests for each point of the graph). Clearly, **Find_large_subset** outperforms the pocket algorithm. The fact that it also outperforms **mult_IncLP** provides strong numerical evidence of the importance of removing bad sequences of examples when they are found. This is the basic difference between **Find_large_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_subset**. This shows that **mult_IncLP** is much less reliable and consistent than **Find_large_subset**. Also, we must mention that, for $n = 16$ and for a training set of 300 or more negative examples, **Find_large_subset** returned, on average, an optimal cluster containing half of the total number of negative examples in the training set.

Finally, we conclude by saying that we have tested our approximation algorithm on other non linearly separable functions and have used it in some constructive neural net algorithms [7] with very good results.
Ackowlegments: This work has been supported by NSERC grant: OCG-0042038.

References

[1] Duda R O and Hart P E 1973 Pattern Classification and Scene Analysis (New-York: Wiley)

[2] Frean M 1990 The Upstart Algorithm: A Method for Constructing and Training Neural Networks *Neural Computation* **2** 198

[3] Gallant S I 1990 Perceptron-Based Learning Algorithms *IEEE Trans. Neural Networks* **1** 179

[4] Golea M and Marchand M 1990 A Growth Algorithm for Neural Network Decision Trees *Europhys. Lett.* **12** 205

[5] Johnson DS and Preparata F P 1978 The Densest Hemisphere Problem *Theor. Comput. Sci.* **6** 93

[6] Karmarkar N 1984 A New Polynomial Time Algorithm for Linear Programming *Combinatorica* **4** 373

[7] Marchand M and Golea M 1993 On Learning Simple Neural Concepts: from Halfspace Intersections to Neural Decision Lists *to appear in Network*.

[8] Mézard M and Nadal J P 1989 Learning in Feedforward Neural Network: the Tiling Algorithm *J. Phys. A* **22** 2191