From: World Congress on Neural Networks: Proceedings of the 1993 Annual Meeting of the International Neural Network Society, vol. 3, pp. 560–563, Hillsdale, NJ: Erlbaum Associates (1993).

A Constructive Algorithm for Neural Decision Lists

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

Abstract

We introduce a new type of neural net architecture that we call *Neural Decision Lists* and present a greedy algorithm for constructing such networks from examples. We first test our algorithm on some "artificially generated" target functions and obtain generalization rates as high as 99% for functions of 50 input variables. We also test the algorithm on data sets taken from a machine learning data base. The good generalization scores obtained on these "real-world" data sets indicate that this architecture is useful in practice.

1 Introduction

When training a feedforward net with backpropagation, one has to deal with two major problems: i) prohibitive training times and ii) having to guess the correct architecture for a given task. To circumvent this, many have recently proposed [4, 11, 3, 2] to use *constructive algorithms*. These are algorithms that construct a feedforward network from examples by training individual perceptrons on some carefully chosen subset of examples. As a consequence, these algorithms generally run much faster than backpropagation. The upshot being that one can (and generally does) assist at 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 exhibit no generalization. As a result, very few good generalization results have been reported for these constructive algorithms. A reason for this failure is that these algorithms are somewhat too general: in order to have some success, a learning algorithm must be biased toward a restricted class of functions (and architectures). For this reason, we propose a constructive algorithm for a restricted class of neural networks that we call *Neural Decision Lists* (NDL). This class is strictly included in (and less general than) the class of *Neural Decision Trees* [4] which, we think, is too general to stand a chance of being learnable. We present numerical results of good generalization scores obtained for this class of networks on both "artificially generated" functions and "real world" data obtained from a machine learning data base.

2 Definitions

Each 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 generalize the notion of *decision lists*, introduced by Rivest [9], to *neural decision lists* (NDL). A NDL (fig. 1) is a list $\mathcal{L}$ of pairs

$$(H_1, v_1), (H_2, v_2), ..., (H_r, v_r)$$

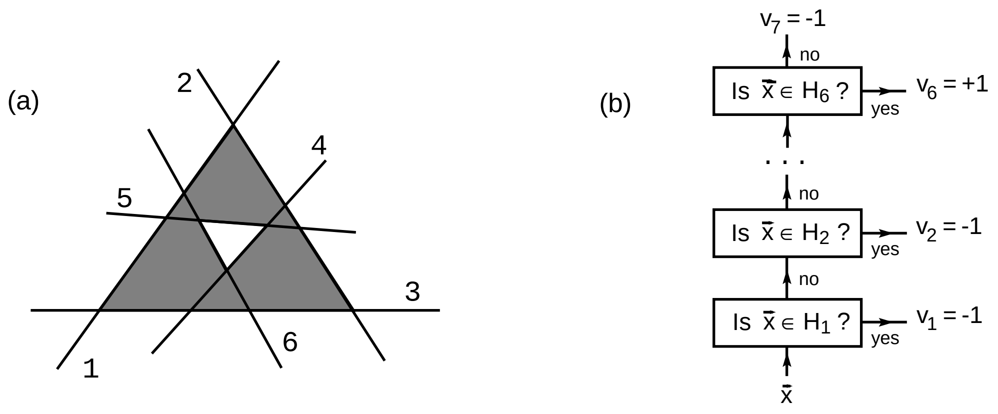

Figure 1: (a)A function to learn; the shaded region represents the set of positive examples. (b) A neural decision list performing this function.

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 $+1$. (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.) This defines a function as follows: for any $\vec{x}$, $\mathcal{L}(\vec{x})$ is defined to be equal to v_j where j is the first (least) index for which $\vec{x} \in H_j$. As in [9], we may think of a NDL as an extended **"if–then–elseif–...else-"** rule (see fig. 1). NDL's can be easily generalized to execute multiple-class functions by allowing each v_j to be multi-valued. Compared to Rivest's decision lists, NDL's have the same structure, but the complexity of the decision allowed at each node is greater. This class of representations, which is a subset of neural decision trees, is strictly richer than halfspace intersections (unions). Moreover, it can be shown[6] that NDL's can be represented as *cascade neural nets* by a suitable choice of inter-nodes connections.

3 Constructing Neural Decision Lists from Examples

The sequential order of the decisions in a NDL suggests that to build such a structure we should first find the halfspace H_1 that covers the most examples of a given class *without* covering the examples of the other class. Then, we only need to build the rest of the NDL on the remaining examples *not covered* by H_1. We then find the halfspace H_2 that covers the most of these (remaining) examples of a given class *without* covering the examples of the other class. We proceed similarly for the other halfspaces until all the examples have been covered. This gives the following greedy heuristic:

1. Let S^+ (S^-) be the set of positive (negative) examples.
2. If $S^+ = \emptyset$, append the pair $(H_r = 1, -1)$ to the decision list and stop.
   Else if $S^- = \emptyset$, append the pair $(H_r = 1, +1)$ to the decision list and stop.
3. 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^+. 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^-.
4. **If** $f^+ > f^-$ **then** append the pair $(H^+, +1)$ to the decision list and remove from S^+ the examples that are covered by H^+. Go to step 2.
   Else append the pair $(H^-, -1)$ to the decision list and remove from S^- the examples that are covered by H^-. Go to step 2.

As discussed in [7], the optimization problem encountered in step 3 is NP-complete. Short of solving this problem in the worst case, we used the approximation algorithm presented in [7] which seems to give good results for *reasonable* distribution of examples. This greedy method may seem to be a naïve one since it is easy to imagine some distribution of examples for which the halfspace consistent with the largest number of examples of a given class 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. But this is not a real setback since our goal is not to find the *minimum* number of halfspaces, but to find a *good* approximation of the target function.

4 Generalization Tests

4.1 Artificially Generated Target Functions

The greedy method was first tested on the *mirror symmetry detector*. Here the target output is +1 if and only if the second half of the n input bits is a mirror reflection of the first half. The greedy always found an intersection two halfspaces. The generalization rate of the greedy algorithm on this function is impressive: an average of 97.5% (over 5 tests) for $n = 50$ after training on only $m = 1600$ examples! 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. We have used $M = 20000$ for $n = 50$. It is important here to test separately on both the positive and negative measure because the positive region accounts only for a fraction of $2^{-n/2}$ of the total measure. The greedy method was also tested on other functions with outstanding generalization rates. For a target function consisting of a intersection of two halfspaces, randomly oriented in the Euclidean region $[-1, +1]^n$, we have obtained a generalization rate of 99% for $n = 50$ after training on 14000 examples. Other results will be reported elsewhere [6].

4.2 Real-World Data Sets

The greedy method was tested on data sets taken from a collection distributed by the machine learning group of the University of California at Irvine (Contact person: Pat Murphy: pmurphy@ics.uci.edu). The results of our algorithm are summarized in table 1. We have included, for comparison, the performance of C4 (a tree induction algorithm [8]) on these data sets as reported by Holte[5]. Also indicated is the "default accuracy"(Def-acc) one has when classifying all the testing examples according to the class containing the largest number of examples. We now comment our results:

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

Table 1: The generalization rate (Gen) of our greedy method, standard deviation (±) and the number of nodes found (k) for various data sets. The training set consists of 2/3 of the m examples and the test set consists of the 1/3 remaining. Each result is averaged over 20 trials. n is the number of input variables and m, the number of examples. Also included, are the results of C4.

CH: *Chess End-Game.* This data set was originally generated and described by Alen Shapiro[10]. To 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[10], 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 does very well on "C4-hand-crafted data".

G2: *Glass identification.* The goal is to determine if a given piece of glass is "float processed" or "non-float-processed". 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. This study was motivated by criminological investigation. At the scene of the crime, the glass left can be used as evidence…if it is correctly identified! The performance of our greedy method was only slightly better (but not significantly) than C4. We think that we do not have enough examples to achieve a higher accuracy.

IR *Iris data set.* This data set (see [1]) contains 3 classes of 50 instances each. Each class refers to a type of iris plant: virginica, versicolor or setosa. One class (setosa) is linearly separable from the other 2; 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 better (but not significantly) than C4.

V0-V1: *United States Congressional Voting Records.* The V0 data set includes votes for each of the U.S. House of Representatives Congressmen on 16 key votes (Salvador aid, education spending,…). 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 congressmen 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 (but not significantly) than that of C4.

Hence, the greedy method for constructing NDL's 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 NDL's and by using a cross-validation data set and other engineering tricks that have proven useful in the past. We are investigating these possibilities. *Ackowlegments:* Work 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] 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*

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

[7] Marchand M and Golea M 1993 An Approximation Algorithm to Find the Largest Linearly Separable Subset of Training Examples *Submitted to WCNN'93.*

[8] Quinlan J R 1986 Induction of Decision Trees *Machine Learning* **1** 81

[9] Rivest R L 1987 Learning Decision Lists *Machine Learning* **2** 229.

[10] Shapiro A D 1987 *Structured Induction and Expert Systems* (Wokingham, UK: Addison-Wesley)

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