Fast Training Algorithms for Multi-layer Neural Nets

Richard P. Brent, Fellow, IEEE

Abstract

Training a multilayer neural net by back-propagation is slow and requires arbitrary choices regarding the number of hidden units and layers. This paper describes an algorithm which is much faster than back-propagation and for which it is not necessary to specify the number of hidden units in advance. The relationship with other fast pattern recognition algorithms, such as algorithms based on k-d trees, is mentioned. The algorithm has been implemented and tested on artificial problems such as the parity problem and on real problems arising in speech recognition. Experimental results, including training times and recognition accuracy, are given. Generally, the algorithm achieves accuracy as good as or better than nets trained using back-propagation, and the training process is much faster than back-propagation. Accuracy is comparable to that for the "nearest neighbour" algorithm, which is slower and requires more storage space.

1. Introduction

This paper is concerned with the problem of training a multi-layer feed-forward neural net, also known as a multi-layer perceptron [1, 2]. "Back-propagation" [3] is a popular training algorithm, but it is slow and requires arbitrary choices regarding the number of hidden units and layers. This paper describes an algorithm which is much faster than back-propagation and for which it is not necessary to specify the number of hidden units in advance. The key idea is to construct a decision tree and then simulate the decision tree with a neural net.

In Section 2 we show how a neural net may be derived from a decision tree. Then, in Section 3, we suggest how to construct a suitable decision tree. An analysis of the complexity of the construction is given in Section 4. Section 5 is concerned with various refinements which improve the performance of the neural net (or decision tree) and reduce the training time. Test results are described in Section 6, and some conclusions are summarized in Section 7.

Decision tree classifiers have been used successfully in many pattern classification applications [4-7]. The idea of constructing a decision tree and using this tree to obtain a neural net is by no means new – see for example [8, 9]. The main contributions of this paper are the complexity analysis of Section 4 and the practical refinements of Section 5.

R. P. Brent is with the Computer Sciences Laboratory, Australian National University, Canberra, ACT 2601, Australia (e-mail: `rpb@cslab.anu.edu.au`).

Appeared in *IEEE Transactions on Neural Networks* 2 (1991), 346–354.

 rpb119 typeset using TEX

A reader who is unfamiliar with decision trees and their relationship to neural networks is advised to read one of the recent surveys [2, 4, 8] before proceeding with Sections 2 and 3.

2. Construction of a Neural Net from a Decision Tree

Suppose the data D is a set of points $x \in R^n$, with each point x assigned a class $k(x)$ from a finite set C of possible classes. Without loss of generality we can assume that $C \subseteq \{1, \ldots, K\}$, where K is an upper bound on the number of classes. A set $S \subseteq D$ of points is used for training, and the aim is to be able to predict the class $k(x)$ of a point x which is not in the training set (i.e. $x \in D \backslash S$). We assume that $k : D \to C$ is a single-valued function.

We restrict our attention to three-layer nets with two hidden layers and one output layer (the input layer is not counted). The nets have n input units and at most K output units. Such a net can form arbitrarily complex decision regions [4]. In fact, it has recently been shown that one hidden layer would suffice [10-12], but it is doubtful if this interesting theoretical result is of practical value (see Theorem B of [13], and also [14]).

Using the training set S, we may construct a *decision tree* T which is a binary tree whose nonterminal nodes correspond to hyperplane tests of the form

$$a_0 + \sum_{j=1}^{n} a_j x_j > 0 \ ? \tag{1}$$

and whose leaves correspond to classes. Such a tree correctly classifies all points in the training set and effectively partitions R^n into a set of convex regions bounded by hyperplanes, with each region V associated with one leaf of T. For an example see Figure 1, where R^2 has been split into five regions numbered $1, 2, \ldots, 5$ by four lines; and Figure 2, which illustrates a corresponding decision tree with $t = 4$ nonterminal nodes.†

Each region V contains training points in at most one class. (If the data is noisy then it may be desirable to *prune* the tree by merging some regions V – see Section 5.1. For the time being we ignore this possibility.) Suppose that T has t nonterminal nodes and $t + 1$ leaves. The value of t is not an input parameter, but is determined by the algorithm which constructs the decision tree, discussed in detail in Section 3.

There is a natural correspondence between the decision tree T and a neural net N which has t units in the first hidden layer, $t+1$ units in the second hidden layer, at most K output units, and hard-limiting (step function) threshold functions. Units in the first hidden layer correspond directly to nonterminal nodes of T. Each such unit evaluates one test of the form (1) with weights $a_1, \ldots, a_n$ and threshold $-a_0$. Each unit in the second hidden layer corresponds to a path from the root of T to a leaf, and effectively computes a conjunction of its inputs or their negations (depending on whether the path is from a node to a right or left child). Thus the weights in these units are 0 or ± 1. Each unit in the output layer corresponds to a single output class and effectively computes a disjunction of its inputs. Thus the weights in these units are 0 or 1.

The neural net N may be used as a good starting approximation for a more conventional training algorithm such as back-propagation, or may be implemented

† Figures are at the end of the paper.

directly in parallel hardware. On a serial computer it is more efficient to use the decision tree T rather than the associated neural net N because the classification of a point x only depends on the results of tests on a path from the root of T to one leaf (the leaf corresponding to the region containing x).

3. Construction of a Decision Tree

Our approach is to find a hyperplane H which splits the training set S into two sets S_0 and S_1 in an optimal way. Here "optimal" is determined by some criterion. We then apply the same criterion recursively to S_0 and S_1, so far as necessary, to construct a decision tree T which correctly classifies all points in S. There are many reasonable criteria for the optimal H [5, 6, 15-17]. Our preferred criterion is to maximize

$$\log \left(\frac{\prod_{k=1}^{K} m_{0,k}! m_{1,k}!}{\left(\sum_{k=1}^{K} m_{0,k}\right)! \left(\sum_{k=1}^{K} m_{1,k}\right)!} \right) , \tag{2}$$

where $m_{i,k}$ is the number of training points of class k in S_i, and logarithms may be taken to any fixed positive base, e.g. 2. We sketch how (2) is derived using considerations of entropy.

Let m_k be the number of training points of class k, and $m = \sum_{k=1}^{K} m_k$ be the total number of training points. The number of ways in which the m training points could be labelled with the class labels $1, \ldots, K$, subject to the constraint that there are m_1 points labelled 1, m_2 points labelled 2, ..., m_K points labelled K, is

$$w = \frac{m!}{m_1! m_2! \cdots m_K!},$$

and the number of bits required to specify such a labelling is $\log_2 w$. Define

$$E(T) = \log w = \log m! - \sum_{k=1}^{K} \log m_k! \,. \tag{3}$$

$E(T)$ may be thought of as the *entropy* associated with T (see (6) below). Any subtree U of T is associated with a subset of the training set S, and $E(U)$ may be defined in the same manner as $E(T)$, with the appropriate modifications to the definitions of m and m_k.

If the hyperplane H splits T into left and right subtrees T_0 and T_1, we define

$$I(H) = E(T) - E(T_0) - E(T_1) \,. \tag{4}$$

$I(H)$ may be thought of as the *information* associated with H. Let the hyperplanes associated with the t nonterminal nodes of T be $H_1, \ldots, H_t$, where $H_1 = H$ is associated with the root of T. Since $E(U) = 0$ for any leaf U, we have

$$E(T) = \sum_{\tau=1}^{t} I(H_\tau) \,. \tag{5}$$

$E(T)$ is determined by the training set, so in order to minimize the size of the tree T we need to maximize the mean value of $I(H_\tau)$. This is a difficult problem, but the "greedy" top-down approach is simply to maximize $I(H_1)$ and then recursively apply the same criterion to the left and right subtrees of T. Since m and m_k (for $k = 1, \ldots, K$) are fixed, maximizing $I(H)$ is equivalent to maximizing (2).

Note that, if we write $p_k = m_k/m$ and use Stirling's approximation in (3), we obtain

$$\frac{E(T)}{m} = -\sum_{k=1}^{K} p_k \log p_k + O\left(\frac{\log m}{m}\right) . \tag{6}$$

as $m \to \infty$ with K fixed. (See Theorem 14.4.1 of [18] for a more precise result.) In view of (6), our criterion (2) is a close approximation to the criterion suggested in Section 2.3 of [5] and also in [16] (which amounts to neglecting the "O" term in (6)). In practice there is probably little to choose between these criteria.

Maximizing (2) is a discrete optimization problem. Since the objective function (2) generally has to be evaluated many times, it is worth precomputing a table of logarithms of integers $1, 2, \ldots, m$. If H is perturbed slightly, as happens during the optimization, it is usually the case that S_0 and S_1 change by only one point, and then (2) can be updated in constant time. Thus, there is no significant advantage to be gained by replacing our criterion by a criterion such as the "Gini index of diversity" [5] purely to reduce the time required to evaluate the criterion.

It may be advantageous to approximate the discrete problem (2) by a continuous problem. For example, suppose that the hyperplane $H = H(a_0, a_1, \ldots, a_n)$ is defined by

$$a_0 + \sum_{j=1}^{n} a_j x_j = 0 \tag{7}$$

where the normal $(a_1, \ldots, a_n)^T$ to the hyperplane has unit length (in the Euclidean norm). Then the signed Euclidean distance $d(x, H)$ of a point x from H is just

$$d(x, H) = a_0 + \sum_{j=1}^{n} a_j x_j \tag{8}$$

(where $d \geq 0$ means $x \in S_1$, $d < 0$ means $x \in S_0$). Let $\phi(z)$ be a smooth, monotonic increasing threshold function with range $(0, 1)$, for example

$$\phi(z) = \frac{1 + \tanh(\lambda z)}{2} = \frac{1}{1 + \exp(-2\lambda z)} , \tag{9}$$

where λ is a positive parameter. Then we can approximate $m_{1,j}$ by

$$\mu_{1,j} = \sum_{x \in S, k(x) = j} \phi(d(x, H)) \tag{10}$$

and $m_{0,j}$ by

$$\mu_{0,j} = \sum_{x \in S, k(x) = j} (1 - \phi(d(x, H))) = m_j - \mu_{1,j} . \tag{11}$$

Since $\mu_{i,j}$ is not necessarily an integer, we need to replace $\log z!$ by $\log \Gamma(z + 1)$ (or an approximation to it) in (2).

Note that the continuous approximation tends to the discrete problem as the parameter $\lambda \to \infty$. In practice, optimization becomes more difficult as λ is increased, because the derivatives of the objective function increase with λ.

Once (2) has been approximated by a continuous problem, continuous optimization algorithms [19-21] are applicable. For example, following the custom with back-propagation, we could use steepest ascent. However, steepest ascent generally converges very slowly [21, 22]. Instead, we recommend a suitable version of the conjugate gradient algorithm [23], which can be implemented with little more work than steepest ascent and is usually much faster. (Similar remarks are relevant to back-propagation – see [24-26].) For details, see Section 5.5. More sophisticated optimization algorithms such as quasi-Newton algorithms [20, 27] could also be used, and would probably be faster than conjugate gradients. Our implementation uses conjugate gradients because of its simplicity and low storage requirements.

4. Analysis of Complexity

There are two reasons why our approach is much faster than back-propagation:

1. maximizing (2) is a problem with only n degrees of freedom (the parameters defining the hyperplane H), whereas back-propagation attempts to find at least nt parameters simultaneously, where t is the number of hidden units in the first hidden layer of the neural net.
2. As we recursively construct the decision tree, the relevant part of the training set becomes progressively smaller. Thus only part of the problem (determination of the hyperplane corresponding to the root) involves the whole training set.

To give a rough estimate of the computational work involved, suppose (as above) that there are m points in the training set, each training point has n real coordinates, and there are K classes. Our algorithm generates a tree with t nonterminal nodes, so a net with $2t+1$ hidden units and at most K output units. Assume that $K = O(t)$ so our net has $O(t)$ units. For comparison with back-propagation, assume that the back-propagation net has the same number of units. Since back-propagation is applied to a continuous problem, assume that a continuous approximation to (2) is used for our objective function (although the discrete problem may be solved even faster, at the possible expense of a larger tree – see examples in Section 6).

Evaluating our objective function (12) takes time $O(mn)$, since m inner products need to be computed. The partial derivatives of the objective function may also be computed in time $O(mn)$ via the chain rule. For purposes of comparison, assume that $O(n)$ evaluations of the objective function and its partial derivatives are required to obtain a sufficiently good solution to an optimization problem with n degrees of freedom. This should be true if the conjugate gradient optimization algorithm is used.

Assuming that the tree generated by our algorithm is roughly balanced (i.e. $|S_0| \sim |S_1|$ etc), then for each of $O(\log t)$ levels of the tree the work required is $O(mn^2)$, and the total work is $O(mn^2 \log t)$. The assumption of a balanced tree is reasonable since the objective function encourages the growth of a well-balanced tree.

By way of comparison, back-propagation involves one optimization problem with $\Omega(nt)$ degrees of freedom, so requires work $\Omega(mn^2t^2)$ under the same assumptions. We conclude that our training algorithm should be faster by a factor of order $t^2/\log t$.

Although the trees generated by our algorithm in numerical tests (Section 6) have invariably been well balanced, we can not prove that this is always true. If an unbalanced tree does occur it has at most $O(t)$ levels, and for each level the work required is $O(mn^2)$, so the total work is $O(mn^2t)$. Thus, even in this extreme case our algorithm should be faster than back-propagation.

The storage requirements of our algorithm are no worse than those of the back-propagation algorithm. If steepest ascent is used, both algorithms require space $O(mn)$ for the training points and $O(nt)$ for the tree or net generated. If quasi-Newton optimization algorithm are used, our algorithm has lower intermediate storage requirements ($O(n^2)$ versus $O(n^2t^2)$).

If the neural net is to be simulated on a serial computer, processing one input requires $O(nt)$ operations, but this can be reduced to $O(n \log t)$ by using the equivalent decision tree (assuming as above that the tree is well-balanced so its depth is $O(\log t)$). The nearest-neighbour algorithm (used as a benchmark in Section 6) takes $O(mn)$ operations when implemented in a straight-forward way. In [28] it is suggested that this can be reduced to $O(c_n \log m)$ (on average) using k-d trees [29], but this is only relevant for small n as c_n appears to increase exponentially with n.

The "conditional class entropy" approach of [17] is similar to ours at the top level, i.e. construction of the first hyperplane H_1. However, for subsequent hyperplanes the approach of [17] differs from ours because it involves the estimation of $O(K2^t)$ conditional probabilities to construct t hyperplanes. Thus, the approach of [17] is probably slower than ours if t is large.

5. Refinements

5.1 Pruning

If the training set S is large and the data noisy, there may be regions where training points with several values of $k(x)$ are distributed in a manner determined mainly by the noise. It is pointless to attempt to subdivide such regions as this will only increase the size of the decision tree T without improving its accuracy on points $x \in D \backslash S$. Unfortunately, it is not always easy to recognise when this situation has arisen. The approach recommended in [5] is to generate a tree T which may be too large, then *prune* it by merging sets of neighbouring nodes if it appears that T is larger than warranted by the data. The criteria for merging and for terminating the pruning process depend on estimates of the misclassification error before and after pruning. For details we refer to Chapter 3 of [5], and also [8, 30]. The results presented in Section 6 do not make use of pruning.

5.2 Dimension Reduction

It is desirable for the dimension n of the data space to be kept as low as possible. One reason is the "curse of dimensionality" [31]. Suppose the data space D is the unit cube $[0, 1]^n$. To sample D with m equally spaced training points on a grid with spacing h requires $m = (1/h)^n$. If $h = 0.1$, this may be feasible for $n \leq 3$, but it is certainly not feasible for $n \geq 10$. Another reason is that for $m \leq n$ it is possible to find a hyperplane H on which *all* the training points lie; by perturbing H slightly we can usually separate the classes of the training points by hyperplanes which are quite useless for classifying points outside the training set. Finally, the complexity analysis (Section 4) shows that the training time can be expected to increase rapidly with n.

For these reasons, it is often desirable to reduce the dimension n of the data space by a preliminary transformation of the data. One way to do this is to perform a principal components analysis [32, 33]. Let A be the overall scatter matrix for the training data, and suppose that $A = Q^T \Lambda Q$, where Λ is a diagonal matrix of eigenvalues $\lambda_1 \geq \cdots \geq \lambda_n$ and Q is an orthogonal matrix of eigenvectors. If $\lambda_{\nu+1}^2$ is negligible compared to λ_1^2, then we may reduce the dimension to ν by making the linear transformation $y = Q^T x$ and discarding the coordinates $y_{\nu+1}, \ldots, y_n$. (Numerically, it may be preferable to use the singular value decomposition [34].)

The need for dimension reduction often arises when the data is a discretization of continuous data. Examples are given in Section 6.

5.3 The Discrete Optimization Problem

Let $F(a_0, a_1, \ldots, a_n)$ be the discrete objective function (2) or some similar objective function. Here $a_0, a_1, \ldots, a_n$ are the coefficients defining a hyperplane H as in (7). Finding the global maximum of $F(a_0, a_1, \ldots, a_n)$ is generally too difficult. We recommend the following alternative –

1. Fix $(a_1, \ldots, a_n)$, which specifies the normal to H, using one or more of the heuristics C, R and/or U described below.
2. Compute the inner products $V(x) = \sum_{j=1}^{n} a_j x_j$ occurring in (8) for each of the m training points x. This takes time $O(mn)$.
3. Sort the m values $V(x)$. This takes time $O(m \log m)$.
4. Solve a 1-dimensional optimization problem to maximize

$$f(a_0) = F(a_0, a_1, \ldots, a_n)$$

   with $a_1, \ldots, a_n$ fixed and only a_0 allowed to vary. Note that $f(a_0)$ is a piecewise constant function with jumps at the m values $-V(x)$. Using the sorted list of $V(x)$ values, we can find the interval on which $f(a_0)$ is maximal in time $O(m)$.
5. Repeat steps 1 – 4 as often as desired, and use the best result obtained. Each iteration of steps 1 – 4 takes time $O(mn) + O(m \log m)$. In practice the term $O(m \log m)$ is usually negligible.

We have used three different heuristics to choose the parameters $(a_1, \ldots, a_n)$ in step 1 above –

C. Try the $K(K-1)/2$ possible normalised differences of centroids of two distinct classes (selected from the K possible classes).

R. Try a random vector of unit length.

U. Try the n possible unit vectors $(1, 0, \ldots, 0)^T, \ldots, (0, \ldots, 0, 1)^T$.

Restriction to heuristic U, as in Quinlan's ID3 system [6], would essentially reduce our decision tree to a k-d tree [28, 29]. This would make hyperplane tests cheaper, but would usually increase the size of the decision tree and decrease the generalization ability [35].

For badly scaled problems it may pay to perform a linear transformation of the data before attempting heuristics C and/or R. For example, we may compute the within-class scatter matrix S_W (see Chapter 4 of [32]), find its Cholesky factorization

$$S_W = LL^T ,$$

(where L is a lower triangular matrix, and we assume that S_W is positive definite or add a small multiple of the identity matrix to ensure this), and then transform data points $x \to y$ by solving $Ly = x$. This transformation reduces the within-class scatter matrix to the identity matrix.

5.4 The Continuous Optimization Problem

From Section 3, the continuous objective function is

$$F(y) = \sum_{i=0}^{1} \left(\sum_{k=1}^{K} G(\mu_{i,k}) - G\left(\sum_{k=1}^{K} \mu_{i,k} \right) \right) , \tag{12}$$

where $G(z) = \log \Gamma(z+1)$ and $y = (y_0, \ldots, y_n)^T$ is a vector of dimension $n+1$ defining a hyperplane $H(a_0, \ldots, a_n)$ as in (7). To obtain the parameters $a_0, \ldots, a_n$ above we normalise $(y_1, \ldots, y_n)$, i.e.

$$a_j = y_j / (y_1^2 + \cdots + y_n^2)^{1/2} \quad (\text{for } j = 0, \ldots, n) . \tag{13}$$

It is desirable to keep $(a_1, \ldots, a_n)$ normalised in order to maintain control over the parameter λ occuring in the threshold function (9). From (8) – (11) we see that multiplying $(a_0, \ldots, a_n)$ by a positive constant c has the same effect as dividing λ by c. Thus, the optimization procedure can in effect impose undesired changes in λ unless we enforce a normalization condition such as

$$a_1^2 + \cdots + a_n^2 = 1 . \tag{14}$$

The partial derivatives of $F(y)$ with respect to y_j may be evaluated in a straightforward manner by use of the chain rule. This is the same as for backpropagation, except that the objective function (12) differs from the sum of squares used in backpropagation. We avoid using partial derivatives with respect to a_j because the expressions for them are more complicated.

When using optimization algorithms, normalization can be performed at the end of each iteration. It may also be worthwhile to ensure by projection that any search directions are orthogonal to $(0, y_1, \ldots, y_n)^T$. This maintains normality up to first order.

If our aim is to approximate the discrete problem, then we may solve a sequence of continuous problems with increasing values of λ. Note that $1/\lambda$ is analogous to the "temperature" parameter used in simulated annealing. Extrapolation methods may be used if we assume that the solutions to the continous problem are given by a power series expansion in $1/\lambda$. (This is similar to the use of penalty functions in constrained optimization [21].) We generally use the heuristic methods described in Section 5.3 to obtain a starting approximation for continuous optimization, and only accept the final result of the continous optimization (or sequence of optimizations) if the discrete objective function is improved.

5.5 The Conjugate Gradient Algorithm

In this subsection, subscripts denote iterations rather than components of vectors. We assume that the objective function $F(y)$ defined in Section 5.4 is to be maximized (not minimized, so some signs change in the standard formulas). This section also applies to backpropagation with the appropriate change in the objective function and increase in the dimension of y.

The basic conjugate gradient iteration of Fletcher and Reeves [23, 36] has the form

$$y_{i+1} = y_i + \lambda_i d_i , \tag{15}$$

where λ_i is chosen to maximize $f(\lambda) = F(y_i + \lambda d_i)$, g_i is the gradient of the objective function F at y_i,

$$d_{i+1} = g_{i+1} + \mu_i d_i \ , \tag{16}$$

and

$$\mu_i = \frac{g_{i+1}^T g_{i+1}}{g_i^T g_i} \ , \tag{17}$$

except that $\mu_i = 0$ for $i = 0$ and whenever the algorithm is restarted (see below).

Observe that the standard back-propagation iteration using steepest ascent has the same form as (15) – (16), except that for back-propagation we would take λ_i and μ_i to be constants defining the strength of the "reinforcement" and "momentum" terms. Thus the work involved per iteration is almost the same for steepest ascent as for conjugate gradients. There is convincing theoretical and empirical evidence that conjugate gradients should converge faster (i.e. require less iterations) than steepest ascent.

Maximizing $f(\lambda)$ is a 1-dimensional optimization problem which may be solved using an efficient algorithm such as *localmin* of [19]. As for the discrete problem (Section 5.3), it is useful to precompute inner products of the search direction d_i and the points x in the training set. This requires time $O(mn)$ and once it is done each evaluation of $f(\lambda)$ requires time only $O(m)$. Thus, there is little point in trying to save time by skimping on accurate 1-dimensional optimization.

Rather than trying to maximize $f(\lambda)$, we may try to find a zero of the derivative $f'(\lambda)$ using an algorithm such as *zero* of [19]. This is preferable numerically (provided a check is made that the algorithm has found a local maximum and not a minimum). The work involved is about the same as for maximizing $f(\lambda)$ directly, provided the appropriate inner products are precomputed.

There are several different formulations of the conjugate gradient algorithm. Some of these are equivalent on quadratic functions, but not on general functions [21, 37]. Shanno [38] suggests some conjugate gradient methods which do not depend on exact line searches. However, in this particular application the line search is inexpensive if performed as described above. We use the iteration (15) – (17) because of its simplicity and the fact that the denominator in (17) can not vanish unless $g_i = 0$.

It is important to note that the conjugate gradient algorithm must be restarted periodically in order to guarantee superlinear convergence [39]. The usual recommendation [21] is to restart after $\dim(y) + 1$ iterations. Thus, we set $\mu_i = 0$ whenever i is divisible by $n + 2$.

6. Test Results

Our algorithm has been tested on artificial problems such as the parity problem; on real problems arising in speech recognition [40-42]; and on a variety of other problems from various sources [43, 44]. The algorithm has been implemented in Pascal on a VAX 11/750 (under VMS) and in Turbo Pascal on an IBM PC.

Comparisons of training algorithms often use the number of "epochs" (passes through the training data) as a machine-independent measure of training time, but this is only appropriate when the algorithms being compared involve similar amounts of work per epoch. This is not the case when comparing our algorithm with back-propagation. Thus, the training times given below are the (machine-dependent) CPU times required for training on a VAX 11/750. The ratios of these times should be similar on other serial computers with floating-point hardware, but could be significantly different on parallel computers.

When conjugate gradients or steepest ascent is used we solve a sequence of problems with increasing λ (typically $\lambda = 1, 2, 4, 8, 16$) as suggested in Section 5.4. As a benchmark we use the "nearest neighbour" algorithm, which simply classifies a point x according to the class of the nearest point in the training set. Here "nearest" depends on the norm used – unless otherwise specified it is the Euclidean norm.

The concept of linear separability [45, 46] is clear if there are only two classes; in this case data is linearly separable if there is a hyperplane H such that all points in class 1 lie on one side of H and all points in class 2 lie on the other side of H. If the number of classes is $K > 2$, several definitions are possible. We say that data is *linearly separable* if there exists a decision tree with at most K leaves (so at most $K - 1$ hyperplanes) which classifies the data correctly.

6.1 The Parity Problem

The parity problem is to determine the parity (i.e. sum mod 2) of n-vectors consisting of zeros and ones. Clearly n hyperplanes $\sum_{i=1}^{n} x_i = j - \frac{1}{2}$ $(j = 1, \ldots, n)$ could be used to separate the data. Thus, a decision tree with n nonterminal nodes and $n + 1$ leaves certainly exists. Our program was able to find such a tree for $n \leq 6$. For larger n the program could find a tree with rather more than n nonterminal nodes.

For example, with dimension $n = 10$ and $m = 523$ training points (approximately 50% of the 1024 possible points), the best result was a failure rate of 3.2% after training for 4729 seconds using steepest ascent. The decision tree generated had $t = 19$ nonterminal nodes, so the neural net derived from it had $2t + 1 = 39$ hidden units. The conjugate gradient algorithm was faster than steepest ascent but gave slightly worse recognition accuracy. This was due to minor differences in the stopping criteria for the optimization algorithms rather than any inherent property of conjugate gradients versus steepest ascent. The discrete algorithm was significantly faster but also significantly less accurate. In fact, its performance on points x which were not in the training set was no better than random. The nearest neighbour algorithm is hopeless (for the parity problem) since it usually gives the wrong answer for points x which are not in the training set! Our results for $n = 10$ are summarized in Table 1. Similar results for $n = 12$ and $m = 1049$ are given in Table 2. The error rates are measured over a random sample of all possible x, including points in the training set.

Table 1: Results for the Parity Problem with dimension 10

Method	*Training time (sec)*	*Error Rate (%)*	*Nonterminal nodes*
Discrete	277	26.5	78
Conjugate gradients	2057	6.4	30
Steepest ascent	4729	3.2	19
Nearest neighbour		48.1	

Table 2: Results for the Parity Problem with dimension 12

Method	*Training time (sec)*	*Error Rate (%)*	*Nonterminal nodes*
Discrete	1025	37.8	150
Conjugate gradients	5625	20.9	56
Steepest ascent	11649	12.2	31
Nearest neighbour		72.2	

Comparison with back-propagation is difficult because we have not found any published results for back-propagation with $n \geq 10$ – most published results are for $n \leq 4$, which is trivial for our algorithm (training time less than 2 seconds). Also, most published results are for training with $m = 2^n$, i.e. all points in the input space are included in the training set. This makes the problem much easier as no "generalization" is required.

6.2 Michalski's Soybean Disease Data

This data was distributed by Schwartz [44] as a test for supervised learning programs. It is apparently based on Michalski's "well-known" soybean disease data set. The data is believed to be linearly separable, although our program failed to verify this.

The data has dimension $n = 50$ and each component is an integer in the range $[0, \ldots, 10]$. There are $K = 17$ classes. The training set S has $m = 184$ points (there are two duplicates in the set of 186 points distributed) and the testing set has 102 points.

We applied our discrete algorithm as described in Section 5.3. For each nonterminal node in the decision tree we performed 1-dimensional optimizations with 50 unit vectors (U), 20 random vectors (R) and $K(K-1)/2 = 136$ differences of centroids (C). For vectors (R) and (C) we applied a linear transformation to reduce the within-class scatter matrix S_W to the identity (see Section 5.3). The result was a decision tree with 20 nonterminal nodes and 21 leaves. Using the method described in Section 2, the tree was converted into a neural net with 50 trivial input units, 20 units in the first hidden layer, 21 units in the second hidden layer, and 17 output units. (In fact, 13 of the output units were trivial, so it would be easy to convert the tree into a net with only 28 nontrivial hidden units and 17 nontrivial output units.) The total training time was 559 seconds.

When applied to the test data, there were 5 errors out of 102 trials (i.e. error rate 4.9%). This is better than the error rate of 7.8% obtained with the nearest-neighbour algorithm (using both the Euclidean and L_1 norms), although the difference is not statistically significant.

6.3 The Canberra Speech Data

This data encodes the vowel nuclei from 10 hVd words (e.g. "had", "hid") each spoken once by 15 male Australian speakers [41]. Each vowel nucleus is represented by 84 real numbers (12 cepstral coefficients, derived by 12th order auto-regressive analysis, for each of 7 time frames). Thus $n = 84$ and $K = 10$. Our program discovered that the data is linearly separable (a surprise). We randomly selected 80% of the data for training (so $m = 120$) and used the other 20% for testing.

Applying the discrete algorithm with $n = 84$, we obtained a tree with 9 nonterminal nodes in 1011 seconds, and the error rate was 42%. Better results were obtained after using the procedure described in Section 5.2 to reduce the dimension of the data. With $n = 15$ the training time was reduced to 364 seconds and the error rate was reduced to 16%. The number of nonterminal nodes was unchanged. Further reduction of n reduced the training time (as expected from the complexity analysis) but increased the number of nonterminal nodes and the error rate. For example, with $n = 4$ the training time was 333 seconds, the number of nonterminal nodes was 24, and the error rate was 42%.

Dimension reduction did not improve the performance of the nearest-neighbour algorithm. For $n = 84, 15$ and 4 the nearest-neighbour algorithm gave error rates of 23%, 29% and 42% respectively.

The continuous approximation (Sections 5.4 – 5.5) generally gave similar results to the discrete algorithm, but the training time was increased by a factor of 2 to 4 for conjugate gradients and 6 to 12 for steepest ascent. This was still much faster than back-propagation, which required several hours to give error rates in the range 23% to 30%.

6.4 The Deterding Speech Data

This data [40, 43] encodes 11 vowels each spoken 6 times by 15 British speakers (both male and female). 8 speakers (4 male and 4 female) were used for training, and a different 7 speakers (4 male and 3 female) were used for testing. The front end processing is described in [42] (excerpt available in [43]). The result is 10 real numbers per utterance. Thus $n = 10$, $K = 11$ and $m = 528$.

The problem is more difficult than that described in Section 6.3 because of the use of different speakers (of both sexes) for training and testing. Our method performed best with dimension reduction to $n = 8$. Using the conjugate gradient algorithm, it generated a decision tree with 48 nonterminal nodes in 2511 seconds. Results using single-layer and multi-layer perceptrons and the nearest neighbour algorithm (as well as some other algorithms omitted here) are given by Robinson in [42, 43]. The training times for the multi-layer perceptrons are not known, but Robinson states "I ran it on several MicroVax II's for many nights", so our method is certainly much faster. The results (from Robinson except for our method) are summarized in Table 3.

Table 3: Results for the Deterding Speech Data

Method	*Error Rate (%)*	*Hidden Units*
Single-layer perceptron	67	
Multi-layer perceptron	49	88
	55	22
	56	11
Nearest neighbour	44	
Our method ($n = 10$)	53	81
Our method ($n = 8$)	42	97

It is interesting that the nearest neighbour algorithm is the best of those considered by Robinson. Our method performs slightly better than multi-layer perceptrons trained by back-propagation (using a comparable number of hidden units), and about as well as the nearest neighbour algorithm.

7. Conclusion

We have shown how to "train" a multi-layer perceptron by first constructing a decision tree and then deriving the perceptron from the decision tree. Generally our algorithm achieves recognition accuracy as good as or better than multi-layer perceptrons trained using back-propagation, and the training process is much faster than back-propagation. This is true even if various "ad hoc" modifications [47-49] are made to speed up the convergence of back-propagation. The recognition accuracy achieved by our algorithm is comparable to that for the nearest neighbour algorithm, which is slower and requires more storage space. We conclude that our algorithm should be useful for practical pattern recognition. It also serves to demonstrate a close connection between neural nets and some classification and data retrieval methods used by statisticians [5, 32, 33, 50], computer scientists [6, 28, 29] and others [2, 51-54]. We do not claim any relevance to the manner in which the human brain processes information [1, 55, 56], but for many applications the performance of the neural net is more important than its plausibility as a biological model.

Acknowledgements

Many thanks to H. T. Kung for interesting me in the topic during a visit to Carnegie-Mellon University; to Zoran Aleksic, George Gusciora, Simon Hawkins, H. T. Kung, J. Bruce Millar, Ah Chung Tsoi and Xin Yao for illuminating discussions, the provision of test data, and assistance in finding references. The referees' comments helped to improve the presentation and bring the references up to date. A preliminary version of this paper was presented at Stanford University in February 1990 as one of the Forsythe memorial lectures [57].

References

1. F. Rosenblatt, *Perceptrons and the Theory of Brain Mechanisms*, Spartan Books, New York, 1962.
2. R. P. Lippmann, An introduction to computing with neural nets, *IEEE ASSP Magazine*, April 1987, 4-22.
3. D. E. Rumelhart, G. E. Hinton and R. J. Williams, Learning internal representations by error propagation, in [54], 318-362.
4. R. P. Lippmann, Pattern classification using neural networks, *IEEE Communications Magazine*, November 1989, 47-50 and 59-64.
5. L. Breiman, J. H. Friedman, R. A. Olshen and C. J. Stone, *Classification and Regression Trees*, Wasdsworth International, Belmont, California, 1984.
6. J. R. Quinlan, Induction of decision trees, *Machine Learning* 1 (1986), 81-106.
7. A. K. Jain, Advances in statistical pattern recognition, *Pattern Recognition Theory and Applications* (edited by P. A. Devijver and J. Kittler), Springer Verlag, New York, 1989, 1-19.
8. I. K. Sethi, Entropy nets: from decision trees to neural networks, *Proc. IEEE* 78 (1990), 1605-1613.
9. P. E. Utgoff, Perceptron trees: a case study in hybrid concept representations, *Connection Science* 1, 4 (1989), 377-391.
10. G. Cybenko, Continuous valued neural networks: approximation theoretic results, in [58], 174-183.
11. K. Funahashi, On the approximate realization of continuous mappings by neural networks, *Neural Networks* 2 (1989), 183-192.
12. A. N. Kolmogorov, On the representation of continuous functions of several variables by superpositions of continuous functions of one continuous variable and addition, *Dokl. Akad. Nauk SSSR* 114 (1957), 679-681. *AMS Translation* 28 (1963), 55-59.
13. G. G. Lorentz, The 13-th problem of Hilbert, *Proc. Symposia in Pure Mathematics Vol. 28: Mathematical Developments arising from Hilbert Problems* (edited by F. Browder), Amer. Math. Soc., 1976, 419-430.
14. G. J. Gibson and C. F. N. Cowan, On the decision regions of multilayer perceptrons, *Proc. IEEE* 78 (1990), 1590-1594.
15. R. L. Harvey, *An introduction to multi-layer perceptrons*, Project Report NN-1, MIT Lincoln Lab., Nov. 1988.
16. C. Koutsougeras and C. A. Papachristou, Training of a neural network for pattern classification based on an entropy measure, *Proc. 1988 IEEE Conference on Neural Networks, Vol. 1*, July 1988, 247-254.
17. M. Bichsel and P. Seitz, Minimum class entropy: a maximum information approach to layered networks, *Neural Networks* 2 (1989), 133-141.

18. R. E. Blahut, *Theory and Practice of Error Control Codes*, Addison-Wesley, Reading, Mass., 1983.
19. R. P. Brent, *Algorithms for Minimization without Derivatives*, Prentice-Hall, New York, 1973.
20. J. Dennis and R. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, New York, 1983.
21. J. Kowalik and M. R. Osborne, *Methods for Unconstrained Optimization Problems*, American Elsevier, New York, 1968.
22. G. E. Forsythe, On the asymptotic directions of the s-dimensional optimum gradient method, *Numer. Math.* 11 (1968), 57-76.
23. R. Fletcher and C. M. Reeves, Function minimization by conjugate gradients, *Computer J.* 7 (1964), 149-154.
24. E. Barnard and R. A. Cole, *A neural-net training program based on conjugate-gradient optimization*, Tech. Report CSE 89-014, Oregon Graduate Center, Beaverton, Oregon, July 1989.
25. R. L. Watrous, Learning algorithms for connectionist networks: applied gradient methods of nonlinear optimization, *Proc. 1987 IEEE Conference on Neural Networks, Vol. 2*, 619-627.
26. P. J. Werbos, Backpropagation: past and future, *Proc. 1988 IEEE Conference on Neural Networks, Vol. 1*, July 1988, 343-353.
27. S. Becker and Y. le Cun, *Improving the convergence of back-propagation learning with second order methods*, Department of Computer Science, University of Toronto, 1987.
28. S. M. Omohundro, Efficient algorithms with neural network behaviour, *Complex Systems* 1 (1987), 273-347.
29. J. L. Bentley, Multidimensional divide and conquer, *Comm. ACM* 23 (1980), 214-229.
30. J. R. Quinlan, Simplifying decision trees, *Int. J. Man-Machine Studies* 27 (1987), 221-234.
31. R. E. Bellman, *Adaptive Control Processes*, Princeton Univ. Press, Princeton, New Jersey, 1961.
32. R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, Wiley, New York, 1973.
33. A. D. Gordon, *Classification – Methods for the Exploratory Analysis of Multivariate Data*, Chapman and Hall, London and New York, 1981.
34. G. H. Golub and C. Van Loan, *Matrix Computations*, Johns Hopkins Press, Baltimore, Maryland, 1983.
35. L. Atlas, R. Cole, Y. Muthusamy, A. Lippman, J. Connor, D. Park, M. El-Sharkawi and R. J. Marks II, A performance comparison of trained multilayer perceptrons and trained classification trees, *Proc. IEEE* 78 (1990), 1614-1619.
36. R. Fletcher, Conjugate direction methods, in *Numerical Methods for Unconstrained Optimization* (edited by W. Murray), Academic Press, London and New York, 1972, 73-86.
37. J. Daniel, *The Approximate Minimization of Functionals*, Prentice-Hall, New York, 1971.
38. D. Shanno, Conjugate gradient methods with inexact line searches, *Math. of Operations Research* 3 (1978), 249-256.
39. H. Crowder and P. Wolfe, Linear convergence of the conjugate gradient method, *IBM J. Research and Development* 16 (1972), 431-433.

40. D. H. Deterding, *Speaker Normalisation for Automatic Speech Recognition*, Ph. D. thesis, Cambridge University, Cambridge, UK, 1989.
41. J. B. Millar, M. O'Kane and P. Bryant, Design, collection, and description of a database of spoken Australian English, *Australian J. of Linguistics* 9 (1989), 165-189.
42. A. J. Robinson, *Dynamic Error Propagation Networks*, Ph. D. thesis, Department of Engineering, Cambridge University, Cambridge, UK, 1989.
43. S. E. Fahlman, *NN-Bench*, Test data available via electronic mail, August 1989.
44. J. Schwartz, Data distributed by electronic mail, December 1988.
45. M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge, Mass., 1969.
46. N. J. Nilsson, *Learning Machines*, McGraw Hill, 1965.
47. S. E. Fahlman, *An empirical study of learning speed in back-propagation networks*, Report CMU-CS-88-162, Dept. of Computer Science, Carnegie-Mellon Univ., June 1988.
48. G. E. Hinton, *Connectionist Learning Procedures*, Tech. Report CMU-CS-87-115 (version 2), Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Dec. 1987.
49. J. J. Hopfield, Learning algorithms and probability distributions in feed-forward and feed-back networks, *Proc. Acad. Sci. USA* 84 (1987), 8429-8433.
50. A. R. Barron and R. L. Barron, Statistical learning networks: a unifying view, in [58], 192-203.
51. Y. le Cun, *A theoretical framework for back-propagation*, Department of Computer Science, University of Toronto, 1987.
52. W. Y. Huang and R. P. Lippmann, Neural net and traditional classifiers, in [55], 387-396.
53. K. M. Passino, M. A. Sartori and P. J. Antsaklis, Neural computing for numeric-to-symbolic conversion in control systems, *IEEE Control Systems Magazine* 9, 3 (April 1989), 44-51.
54. D. E. Rumelhart, J. L. McClelland *et al* (editors), *Parallel Distributed Processing, Vol. 1*, MIT Press, Cambridge, Mass., 1986.
55. D. Z. Anderson (editor), *Neural Information Processing Systems*, Amer. Inst. of Physics, New York, 1988.
56. F. Crick, The recent excitement about neural networks, *Nature* 337, 129-132, Jan. 1989.
57. R. P. Brent, *Fast training algorithms for multi-layer neural nets*, Technical Report NA-90-03, Department of Computer Science, Stanford University, March 1990. Extended abstract in *Proc. First Australian Conference on Neural Networks*, (edited by M. Jabri), Electrical Engineering, University of Sydney, January 1990, 97-98.
58. E. J. Wegman, D. T. Gantz and J. J. Miller (editors), *Computing Science and Statistics, Proceedings of the 20th Symposium on the Interface*, Amer. Statist. Assn., Alexandria, Virginia, 1988.

Richard P. Brent (M'72-SM'83-F'91) received the B.Sc. (hons) degree in mathematics from Monash University, Australia, in 1968, the Ph.D. degree in computer science from Stanford University, Stanford, California, in 1971, and the D.Sc. degree from Monash University in 1981.

From 1971 to 1972 he worked in the Mathematical Sciences Department, IBM Research Center, Yorktown Heights, New York. Since 1972 he has been at the Australian National University, Canberra, Australia, where he is currently Professor of Computer Sciences and Head of the Computer Sciences Laboratory in the Research School of Physical Sciences and Engineering. His research interests include artificial neural networks, parallel computer architectures, analysis of algorithms (especially parallel algorithms), numerical analysis, and computational number theory.

Dr Brent is a member of ACM, AMS, SIAM, Sigma Xi, a fellow of the Australian Academy of Science, and a fellow of the IEEE.

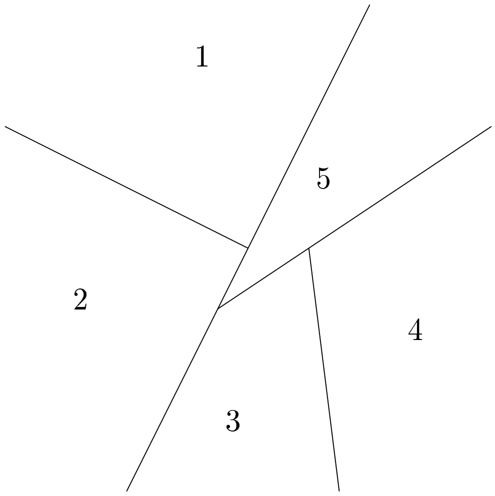

Figure 1: Partition of R^2 into five regions.

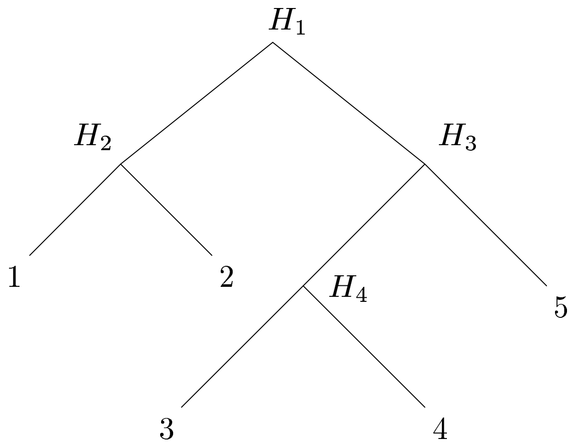

Figure 2: A corresponding decision tree.