The Dependence Identification Neural Network Construction Algorithm

John O. Moody, *Student Member, IEEE,* and Panos J. Antsaklis, *Fellow, IEEE*

Abstract—**An algorithm for constructing and training multilayer neural networks, dependence identification, is presented in this paper. Its distinctive features are that i) it transforms the training problem into a set of quadratic optimization problems that are solved by a number of linear equations, ii) it constructs an appropriate network to meet the training specifications, and iii) the resulting network architecture and weights can be further refined with standard training algorithms, like backpropagation, giving a significant speedup in the development time of the neural network and decreasing the amount of trial and error usually associated with network development.**

I. Introduction

The main tools for training multilayer feedforward neural networks are gradient-based optimization techniques such as the backpropagation (BP) algorithm developed by Rumelhart [19]. Typical gradient descent algorithms are susceptible to local minima, sensitive to initial conditions, and slow to converge. Gradient descent can work quite well with the appropriate set of initial conditions and with a proper network architecture, but using random initial conditions and guessing at the network architecture usually leads to a slow and ponderous training process. BP requires that the nonlinear neural activation functions be restricted to those with continuous derivatives. The designer must specify the number of network layers and the number of neurons in the "hidden layers" when using basic BP. A method of quickly specifying a network architecture and set of initial weight values for possible further training through gradient descent would be extremely valuable.

One method of developing an appropriate network architecture without having to use trial and error is pruning. Pruning techniques start out with a very large network, with the goal being to eliminate superfluous weights and/or neurons. An important (and popular) tool for pruning weights and neurons is a method known as "weight decay," see [8], [9], [13], and [22]. Weight decay causes all of the weights in the neural network to decrease slightly in magnitude at every BP training step. If BP training is constantly emphasizing certain weights, then the slight decay will not hurt these values, however, if a weight is rarely changed by BP, then the decay process will make it go to zero. It is then possible to remove the weights that have near zero magnitude. The rules of weight decay can easily be extended to work equally on all the weights associated with individual neurons, thus allowing the decay to eliminate entire neurons from the network instead of just individual weights. The method of weight decay has two main problems, it requires that the network be very large at the beginning of training and it relies on gradient-based optimization methods. Hardware or time restrictions may interfere with the size requirements for the initial network, and gradient descent has its own set of problems. Pruning methods still require the designer to guess at the number of layers required for the network, though results from [2], [3], [10], and [11] making guessing at the required number of hidden layers a relatively simple task. The algorithm presented in this paper attempts to alleviate the difficulties of constructing a network with pruning techniques, however, the resulting network can still be refined with pruning and iterative training methods.

The most impressive progress in network construction algorithms has been for Boolean networks. Boolean networks realize Boolean functions, with the inputs, outputs, and neural activations all being binary valued. Although our interest is in networks that are not necessarily Boolean, some of the algorithms developed for Boolean networks have features in common with the dependence identification (DI) algorithm presented in Section III. For this reason several methods of constructing Boolean networks are outlined below, details can be found in [6], [8], [14], and [15].

Marchand *et al.* [14] developed a method for constructing a Boolean network with a single hidden layer. The inputs and outputs and hidden-layer activations are all ± 1. The idea is to train a single hidden-layer neuron to give a correct response for all of the patterns that should generate a $+1$ output and at least one of the patterns that generates a -1 response. New hidden units are then added, each one getting one more of the -1 responses correct, while maintaining the correct response for all of the $+1$ patterns. When all of the -1 responses are answered correctly by at least one hidden-layer unit, Marchand *et al.* prove that the perceptron training algorithm [18] can be applied to produce the output layer weights, i.e., they prove that the hidden layer linearly separates the pattern space.

The upstart algorithm developed by Frean in 1990 [6] uses a different kind of network architecture. The first step is to train a single neuron to correctly classify a portion of the patterns. If not all of the patterns are correctly classified, then two new neurons are added to the network. These neurons receive the input patterns as their own input and feed their output to the first neuron. One of these neurons has a positive weight and the other has a negative weight. The two neurons' weights are then used to correct some of the mistakes made by the first neuron. If the first neuron still does not classify all patterns correctly then four new neurons are added, two for each of

Manuscript received September 22, 1993; revised February 10, 1995.

The authors are with the Department of Electrical Engineering, University of Notre Dame, Notre Dame, IN 46556 USA.

Publisher Item Identifier S 1045-9227(96)00486-9.

the secondary neurons, and the process is repeated until the first neuron correctly classifies all of the input–output patterns. Frean proves that each additional layer will eliminate at least one of the incorrectly classified patterns at the output layer, thus the process must eventually halt.

The tiling algorithm was developed by Mézard and Nadal [15] in 1989. The algorithm is based on the idea that if two input patterns are to give rise to different output values, then their hidden-layer representations must also be different. First a single neuron is trained to correctly classify as large a proportion of the pattern space as possible. Extra units are then added until no two input patterns that have opposite output values have the same hidden-layer representation. The process is then restarted, and the hidden-layer patterns are used as inputs to a new layer. Mézard and Nadal prove that a layer will always have at least one fewer neuron than the layer before it, thus the algorithm will always reach a conclusion with a single output that gives the correct response for all patterns.

The DI algorithm is presented in this paper. DI bears some similarities to the Boolean network construction algorithms, however, it is designed to work with continuous training problems, and it uses the concept of linear dependence, instead of the desired Boolean output value, to group patterns together. The algorithm does not share the problems of network pruning techniques because it builds from a small network up to a large one, and because it does not use gradient descent. The algorithm creates a network and set of initial conditions that are suitable for further iterative or on-line adaptive training with gradient techniques such as BP and weight decay.

The single-layer training method of quadratic optimization, a major tool used by the DI algorithm, is explained in the following section. The DI algorithm is then presented in Section III. Section IV gives examples of neural-network construction using DI. Concluding remarks appear in Section V.

II. Quadratic Optimization

Many methods exist [8] for training single-layer neural networks including steepest descent, least mean squares, and the "perceptron training algorithm" (for single-layer classification problems). In [20], Sartori and Antsaklis' proposed quadratic optimization, which transforms the nonlinear training cost function of a single-layer network into a quadratic one. This method reduces the problem to solving a set of linear equations and is a major tool used by the DI algorithm described in the next section. A description of the method is given here.

Individual neurons perform the following function:

$$y = \phi\left(\sum_{i=1}^{m} w_i u_i\right).$$

The neuron receives m inputs, $(u_1, \cdots, u_m)$, and has a single output y. Each input u_i is multiplied by a corresponding weight factor w_i. The output is formed by performing a nonlinear function $\{\phi: \mathbb{R} \to \mathbb{R}\}$ on the sum of the weighted inputs. The function ϕ is generally a sigmoid, but this is not necessary.

A single-layer neural network receives m inputs and produces n outputs with one artificial neuron for each output. The jth output may be expressed as

$$y_j = \phi\left(\sum_{i=1}^{m} w_{ij} u_i\right)$$

where $1 \le j \le n$ and w_{ij} is the weight between the ith input and the jth output. The network may be expressed in matrix form by defining the inputs and outputs as vectors and the weights as a matrix. Then

$$y = \Phi(uW)$$

where

$$y \in \mathbb{R}^{1\times n} \quad u \in \mathbb{R}^{1\times m} \quad W \in \mathbb{R}^{m\times n}$$

and the ijth element of W, the neural weight matrix, is given by w_{ij}. The notation $\Phi(uW)$ means perform the same function ϕ on every element of the vector uW, forming the new vector $\Phi(uW)$

$$\{\Phi: \mathbb{R}^n \to \mathbb{R}^n\} \quad \Phi_i(x) = \phi(x_i). \tag{1}$$

The overall network represents a static nonlinear mapping $\mathbb{R}^m \to \mathbb{R}^n$ from the input to the output. The training goal is to make this mapping approximate, as close as possible, some desired mapping from $\mathbb{R}^m$ to $\mathbb{R}^n$. Suppose there are p input–output patterns that are to be used to train the network, defined as follows

$$(u_i^k, d_j^k) \quad i = 1, \cdots, m \quad j = 1, \cdots, n \quad k = 1, \cdots, p$$

where u_i^k is the value of the ith input for the kth training pattern and d_j^k is the jth desired output for the kth training pattern. The following matrices can then be constructed:

$$U \in \mathbb{R}^{p\times m} \quad D \in \mathbb{R}^{p\times n}$$

where U is the matrix of input patterns and D is the matrix of corresponding desired output patterns. The value of the jth output neuron $(1 \le j \le n)$ to the kth training pattern $(1 \le k \le p)$, y_j^k, is given by

$$y_j^k = \phi\left(\sum_{i=1}^{m} w_{ij} u_i^k\right)$$

or in matrix form, the output $Y \in \mathbb{R}^{p\times n}$ of a single-layer neural network to the input U is

$$Y = \Phi(UW)$$

where $W \in \mathbb{R}^{m\times n}$ is the neural weight matrix and $\Phi_{ij}(X) = \phi(x_{ij})$ as in (1).

The neural-network training problem involves making Y as close an approximation as possible to D. If this "closeness" is defined in the least mean squares sense, then the error of the network is given by

$$e = \sum_{k=1}^{p} \sum_{j=1}^{n} (d_j^k - y_j^k)^2. \tag{2}$$

The training goal is then to find a set of weights W which minimizes the error e. The error equation and training goal can also be expressed with the matrix notation

$$\begin{aligned} e &= \operatorname{trace}\,[(D-Y)^T(D-Y)] \\ &= \operatorname{trace}\,\{[D-\Phi(UW)]^T[D-\Phi(UW)]\}. \end{aligned} \tag{3}$$

And the training problem is to find a set of weights that minimizes e

$$\min_W \ \operatorname{trace}\,\{[D-\Phi(UW)]^T[D-\Phi(UW)]\}. \tag{4}$$

To deal with the nonlinearity of (4) consider defining a new matrix $V \in \mathbb{R}^{p\times n}$ such that

$$\Phi(V) = D. \tag{5}$$

If ϕ is bijective then we can take $v_j^k = \phi^{-1}(d_j^k)$, however, it is not necessary that a one-to-one inverse of ϕ exist. For example, the zero-threshold step function is surjective for the domain of $\mathbb{R}$ and the range $\{0, 1\}$ i.e., it has an infinite number of inputs that will evaluate to one and an infinite number of inputs that will evaluate to zero. V can still be defined by assigning some positive number to v_j^k whenever d_j^k is one and assigning some negative number to v_j^k whenever d_j^k is zero. The numbers assigned may be random or some prechosen constants. With the change of variables in effect, the single-layer neural-network training problem is transformed to

$$\min_W \ \operatorname{trace}[(V-UW)^T(V-UW)] \tag{6}$$

which is equivalent to solving

$$U^TUW = U^TV \tag{7}$$

for W if U^TU is positive definite (full rank).

Of course the real problem to be solved is the nonlinear problem given by (4). Note that if a zero error minimum solution to (6) exists, then a W can be found such that it gives a zero error solution to (4). When a zero error solution to (6) does not exist then the relationship between the two solutions is more subtle. A more complete discussion of the error relationship appears below.

A. Error Analysis

The original single-layer neural-network training problem is to find a W that

$$\min_W F(W) \tag{8}$$

where

$$F(W) = \operatorname{trace}\,[D-\Phi(UW)]^T[D-\Phi(UW)]. \tag{9}$$

Let the error matrix, $\epsilon \in \mathbb{R}^{p\times n}$, associated with (8) at any time be given by

$$\epsilon = D - \Phi(UW). \tag{10}$$

Note that the scalar error measure e in (2) and (3) is $e = \operatorname{trace}\,\epsilon^T\epsilon$. The jkth element of ϵ is the error of the jth output neuron to the kth pattern

$$\epsilon_j^k = d_j^k - \phi\left(\sum_{i=1}^m u_i^k w_{ij}\right)$$

where $1 \le j \le n$ and $1 \le k \le p$.

For the case where the change of variables, $\Phi(V) = D$, has been performed, the problem is defined by (6) and is rewritten in terms of a quadratic cost function $\hat{F}$

$$\min_W \hat{F}(W) \tag{11}$$

where

$$\hat{F}(W) = \operatorname{trace}\,(V-UW)^T(V-UW). \tag{12}$$

The error matrix, $\hat{\epsilon} \in \mathbb{R}^{p\times n}$, associated with (11) at any time is given by

$$\hat{\epsilon} = V - UW. \tag{13}$$

The relation [20] between ϵ and $\hat{\epsilon}$ is

$$\epsilon = \Phi(UW + \hat{\epsilon}) - \Phi(UW). \tag{14}$$

Note that each element of ϵ depends only on the corresponding element of $\hat{\epsilon}$ not on the entire $\hat{\epsilon}$ matrix, i.e., ϵ_j^k is a function of $\hat{\epsilon}_j^k$ only

$$\epsilon_j^k = \phi\left[\left(\sum_{i=1}^m u_i^k w_{ij}\right) + \hat{\epsilon}_j^k\right] - \phi\left(\sum_{i=1}^m u_i^k w_{ij}\right) \tag{15}$$

where $1 \le j \le n$ and $1 \le k \le p$.

Equation (15) shows that for a given W, if $\hat{\epsilon} = 0$ (matrix), then $\epsilon = 0$ as well. Thus if the transformed problem (11) has a zero error solution, then a zero solution to the original problem (8) has been found as well. If the neural activation function is bijective, e.g., it is sigmoid, then for any W, $\epsilon = 0$ if and only if $\hat{\epsilon} = 0$. In general, however, $\hat{\epsilon} \neq 0$. This case is now studied.

In view of (15), an approximate relationship between the two errors can be derived that sheds some light into their relationship. In particular

$$\epsilon_j^k \approx \hat{\epsilon}_j^k \phi'\left(\sum_{i=1}^n u_i^k w_{ij}\right) \tag{16}$$

where $\phi'(x) = d\phi/dx(x)$. The approximation is valid for sufficiently small $\hat{\epsilon}_j^k$ by combining (15) and

$$\phi'\left(\sum_{i=1}^m u_i^k w_{ij}\right) = \lim_{\hat{\epsilon}_j^k \to 0} \frac{\phi\left[\left(\sum_{i=1}^m u_i^k w_{ij}\right) + \hat{\epsilon}_j^k\right] - \phi\left(\sum_{i=1}^m u_i^k w_{ij}\right)}{\hat{\epsilon}_j^k}$$

where $1 \le j \le n$ and $1 \le k \le p$.

It is now clear that for ϵ_j^k to be small $\hat{\epsilon}_j^k$ should be small and/or $\phi'\left(\sum_{i=1}^m u_i^k w_{ij}\right)$ should be small. That is, good solutions to the original nonlinear problem are achieved when the weights w_{ij} are such that $\phi'\left(\sum_{i=1}^m u_i^k w_{ij}\right) \approx 0$, or when, in the case of sigmoids, the weights push the neural inputs into the flat regions of the function. This is of course difficult to guarantee as it is heavily dependent on the training patterns. In an attempt to further shed light on the error relationship it is now of interest to examine the conditions under which the W which minimizes $\hat{F}(W)$ adequately approximates the minimum of the original (nonlinear) problem. Suppose we find a weight matrix W^* which solves (11). The associated error matrix is then

$$\hat{\epsilon}^* = V - UW^*.$$

Under what conditions are the elements of ϵ^*, the nonlinear problem's error matrix from (10) with weight matrix W^*, less than or equal to the elements of ϵ, where ϵ is the error matrix associated with any weight matrix? Using the approximation, (16), we want

$$|\epsilon_j^{k*}| \approx \left|\hat{\epsilon}_j^{k*}\phi'\left(\sum_{i=1}^{m} u_i^k w_{ij}^*\right)\right| \le \left|\hat{\epsilon}_j^k\phi'\left(\sum_{i=1}^{m} u_i^k w_{ij}\right)\right| \approx |\epsilon_j^k|$$

where w_{ij}^* and w_{ij} are the ijth elements of W^* and W. The inequality should hold for all patterns k and output neurons j. The inequality can be viewed in terms of the ratio of the slopes of the output activations compared to the ratio of the magnitudes of the errors

$$\left|\frac{\phi'\left(\sum_{i=1}^{m} u_i^k w_{ij}^*\right)}{\phi'\left(\sum_{i=1}^{m} u_i^k w_{ij}\right)}\right| \le \left|\frac{\hat{\epsilon}_j^k}{\hat{\epsilon}_j^{k*}}\right|. \tag{17}$$

Let α_j^k equal the absolute value of the ratio of the slopes for the jth output neuron and the kth training pattern

$$\alpha_j^k = \left|\frac{\phi'\left(\sum_{i=1}^{m} u_i^k w_{ij}^*\right)}{\phi'\left(\sum_{i=1}^{m} u_i^k w_{ij}\right)}\right|.$$

Then for the weight matrix W^* to produce a minimum in the nonlinear space we need

$$|\hat{\epsilon}_j^{k*}| \le \frac{1}{\alpha_j^k}|\hat{\epsilon}_j^k|.$$

If $\hat{\epsilon}^* = V - UW^* = 0$ then the inequality will obviously hold. If the elements of UW^* are high in magnitude, then they will lie along the flat points of the sigmoid and the derivative will be approximately zero, making $\alpha_j^k \approx 0$. In this case the solution W^* should lie close to the optimal solution of the nonlinear problem. If the elements of UW^* are small in magnitude, then the derivative of the sigmoid will be high and it is likely that W^* lies farther away from the optimal solution to the nonlinear problem. Note that the elements of W^* are not considered by themselves. It is necessary to examine the elements of UW^*. This indicates that the set of training patterns has a strong impact on the value of the solutions that are derived.

Note that if the solution produced by quadratic optimization is not close enough to the true minimum, then the solution from quadratic optimization can be used as a set of initial conditions to further reduce the error using some form of nonlinear gradient descent. The solutions from quadratic optimization will generally lie close to the nonlinear solution, making gradient descent an effective tool for fine tuning the error if a further reduction of the error is necessary. In the Appendix of this paper, the relationship between the errors ϵ and $\hat{\epsilon}$ is illustrated through examples.

TABLE I
THE XOR FUNCTION

u_1	u_2	d
0	0	0
1	0	1
0	1	1
1	1	0

III. Dependence Identification

This section introduces the multilayer neural-network construction method DI. An example of network construction is given as an introduction in Section III-A before the algorithm is formally described in Section III-B. Sections III-C and III-D address specific issues of the algorithm.

A. Network Construction Example

A pseudocode description of the DI algorithm is given in the next subsection. A simple example is given here to help the reader develop a more intuitive understanding of the algorithm, though the reader may omit this subsection without loss of continuity. A neural network is to be constructed that realizes the exclusive or (XOR) Boolean function. The XOR function has two Boolean inputs and a single Boolean output. The desired input–output mapping is given in Table I.

The input patterns are augmented with a vector of constant inputs to act as a bias. The U and D matrices are then

$$U = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad D = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

Because the input–output space is Boolean, it makes sense to define the neural activation function ϕ as the step function, which also has output values of zero and one. We now need to define the V matrix such that $\Phi(V) = D$. As mentioned in Section II, the step function is not bijective, but we can still assign values to V that would give rise to the appropriate values in D. One way to do this is to assign a -1 to V when D contains a 0 and a 1 to V when D contains a 1. Thus

$$V = \begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \end{bmatrix}.$$

If a single-layer neural network with weight matrix $W \in \mathbb{R}^{3\times 1}$ is to realize the function, then we need $UW = V$. A solution to this equation exists if

$$\operatorname{rank}(U) = \operatorname{rank}([U \quad V]) \tag{18}$$

but U is rank 3 and $[U \quad V]$ is rank 4, thus a single-layer neural-network solution does not exist with the given activation function and choice for V. The pattern space is not linearly separable, which is a well-known result. If the XOR function is to be realized with the given nondecreasing activation function, then the network must have a minimum of two layers.

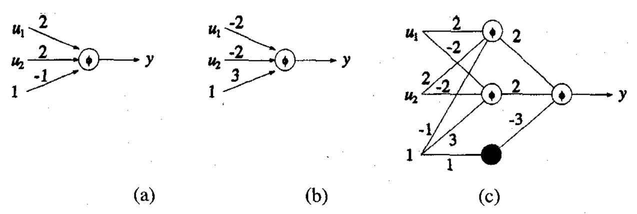


Fig. 1. Components of the XOR neural network.

The rank condition of (18) dictates the entire pattern set cannot be realized at once, but what about a subset? Take the first three rows of U, call them $U_{1:3}$, and the first three rows of V, call them $V_{1:3}$. Now $\operatorname{rank}(U_{1:3}) = \operatorname{rank}([U_{1:3}\ V_{1:3}]) = 3$ so a solution to this problem does exist. An inspection of the three patterns shows that the solution is simply the Boolean OR function. The equation

$$U_{1:3}g_1 = V_{1:3}$$

is solved for g_1 and we obtain

$$g_1 = \begin{bmatrix} 2 \\ 2 \\ -1 \end{bmatrix}.$$

Weight matrix g_1 provides a single-layer neural-network solution to the Boolean OR problem as shown in Fig. 1(a).

Now we take the last three rows of U and V: $U_{2:4}, V_{2:4}$. Once again the two matrices satisfy the rank condition of (18) and we can see that the solution would be the Boolean NAND function. The equation

$$U_{2:4}g_2 = V_{2:4}$$

is solved for g_2

$$g_2 = \begin{bmatrix} -2 \\ -2 \\ 3 \end{bmatrix}.$$

The single-layer solution to the Boolean NAND problem is shown in Fig. 1(b).

The two single-layer neural networks for OR and NAND can be used as building blocks for the multilayer XOR network. First we will define another single-layer network that has an output which never changes. This will act as a bias just like the third column of U. The weight matrix for this bias neuron is then

$$g_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

The first layer weight matrix can then be constructed as

$$W_1 = [g_1\ g_2\ g_3].$$

The output of the hidden layers is then computed

$$\Phi(UW_1) = H = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

and it is easy to see that $\operatorname{rank}(H) = \operatorname{rank}([H\ V]) = 3$, thus a solution $W_2 \in \mathbb{R}^{3\times 1}$ to the final problem $HW_2 = V$ does exist

$$W_2 = \begin{bmatrix} 2 \\ 2 \\ -3 \end{bmatrix}.$$

The network that realizes the XOR problem is shown in Fig. 1(c).

B. The Algorithm

The XOR network in the previous section was constructed using the DI algorithm, which will now be formally defined. A network is to be constructed and trained to approximate a function with m inputs and n outputs. The training set consists of p input–output patterns. The training matrices U and D are the same as defined in Section II. The network is constructed one layer at a time. If hidden layers are necessary, they are created by selecting subsets of the pattern space, as described below, and using the rules for single-layer network construction from Section II. The outputs of the hidden-layer neurons are then used as inputs for the next layer.

To construct the network, first attempt to create a single-layer neural network by solving (6) using the method of quadratic optimization. Compute the error of the single-layer network. If the error is acceptable (less than a given tolerance) then training is complete and the solution is a single-layer network, otherwise create a layer of hidden neurons each of which gets a subset of the complete set of training patterns matched correctly, i.e., every pattern should be classified correctly by at least one hidden-layer neuron. To do this choose an $m \times m$ portion of U (it is assumed that there are more training patterns than there are inputs, i.e., $p > m$.), and the corresponding $m \times n$ portion of D. Use these patterns to solve a single-layer neural-network training problem as above. It is guaranteed that a unique zero error solution to this problem exists if the $m \times m$ portion of U is full rank, see (7) and (18). Repeat this procedure until every pattern is matched by at least one hidden-layer neuron. Patterns in U and the transformed variable V that are correctly matched by a single hidden-layer neuron are linearly dependent, which is the reason for the name DI. The hidden-layer outputs are then treated as inputs to form a new layer. Layers may be added to the network until a maximum number of layers has been added or the network error is within the desired tolerance.

Fig. 2 gives the pseudocode used to implement DI. The inputs to the algorithm include the training patterns U and D as well as the neural activation function ϕ, the maximum number of layers $h_{\max}$, and two tolerances ϵ_N and ϵ_p. The acceptable error for the entire network is given by ϵ_N. The other tolerance, ϵ_p, is used to determine whether the output of the network for an individual pattern is within bounds. Larger values for ϵ_p give fewer hidden-layer neurons.

C. Hidden-Layer Neurons

The DI algorithm constructs a network that solves the single-layer problem within the desired accuracy, if such a

```
Algorithm 1 (Dependence Identification).
input $U \in \mathbb{R}^{p\times m}$, $D \in \mathbb{R}^{p\times n}$,
  $\phi : \mathbb{R} \rightarrow \mathbb{R}$, $\epsilon_N$, $\epsilon_p$, $h_{\max}$.
Create $V$ such that $\Phi(V) = D$
Set $h = 1$
repeat
    Solve $\min_{W_h} \operatorname{trace}((UW_h - V)^T(UW_h - V))$
      $W_h \in \mathbb{R}^{m\times n}$.
    Compute error:
    $e = \operatorname{trace}((\Phi(UW_h) - D)^T(\Phi(UW_h) - D))$
    if ($e > \epsilon_N$) and ($h < h_{\max}$) then
        Set $G$ = empty matrix, $l = 0$
        Set $U_{\text{unmarked}} = U$, $V_{\text{unmarked}} = V$
        repeat
            Choose $u \subset U_{\text{unmarked}}$, $v \subset V_{\text{unmarked}}$
              where $u \in \mathbb{R}^{m\times m}$, $v \in \mathbb{R}^{m\times n}$
            if less than $m$ patterns are
              unmarked then augment $u$ and $v$
              with previously marked patterns.
            Solve:
            $\min_g \operatorname{trace}((ug-v)^T(ug-v))$, $g \in \mathbb{R}^{m\times n}$.
            Add $g$ as new column(s) of $G$.
            $l \leftarrow l + n$
            Mark all patterns which do
              not satisfy $U_{\text{unmarked}}g = V_{\text{unmarked}}$
              within the tolerance $\epsilon_p$.
        until all patterns have been marked.
        Find the hidden layer outputs:
        $H = \Phi(UG) \in \mathbb{R}^{p\times l}$.
        Add a zero column to $G$ with a single
          nonzero element corresponding to the
          constant input.
        $W_h \leftarrow G$
        $U \leftarrow H$
        $m \leftarrow l$
        $h \leftarrow h + 1$
until ($e \leq \epsilon_N$) or ($h = h_{\max}$)
output $W_i$ for $i = 1, \ldots, h$ and the error $e$
```

Fig. 2. Algorithm 1.

solution actually exists. The number of layers created can be bounded with the parameter $h_{\max}$. The number of hidden-layer neurons depends on the number of layers and the desired accuracy. Several authors have shown (with different degrees of ease and generality) that p neurons in the hidden layer suffice to store p arbitrary patterns[1] in a two-layer network (see [1], [17], and [21] for constructive proofs).

The bound on the number of hidden-layer neurons created by DI is proportional to the bound of p described above. Assume that the $m \times m$ portions of U are full rank and that the number of network layers is limited to two (one hidden layer). The U matrix has p rows, so it will be broken into at most ceiling(p/m) linearly dependent groups[2] to form the hidden-layer neurons. This bound is achieved when every group is made of at most m patterns. In practice the number of groups can be much smaller than ceiling(p/m) since larger groups of linearly dependent patterns may be found. Every linearly dependent groups yields n hidden-layer neurons (one for each output). An extra neuron with a constant output is then added to the hidden layer to help form constant offsets and/or thresholds for the output layer. Thus the DI algorithm has an upper bound of ceiling$(p/m)n + 1$ hidden-layer neurons for a two-layer network and assuming the $m \times m$ portions of U are full rank. The DI algorithm assumes that one input is a constant used to form thresholds for the first layer's neurons. Thus for a single-input/single-output network $m = 2$ and $n = 1$ and the bound on the number of hidden-layer neurons created by DI is approximately one half the bound of p.

There may be situations where DI determines a number of neurons and/or layers that may not be physically realizable due to hardware or software constraints. The number of layers may be limited by the parameter $h_{\max}$ in Algorithm 1, but it is not so easy to limit the number of neurons within a hidden layer. One way to decrease the number of hidden-layer neurons is to increase the tolerance ϵ_p. This value is used to determine whether a pattern should be considered part of a linearly dependent set, i.e., if a pattern subset u, d, v and associated weight g satisfies $ug = v$, then another pattern $\hat{u}$, $\hat{d}$, $\hat{v}$ is considered "linearly dependent" if the elements of $|\Phi(\hat{u}g) - \hat{d}|$ are all less than ϵ_p. Linear dependence is written in quotes here because the nonlinear, not the linear, function is actually being used to determine a pattern match. If the linear function were used, then the check would be to see if the elements of $|\hat{u}g - \hat{v}|$ are all less than ϵ_p. Increasing the value of ϵ_p increases the number of patterns that are considered linearly dependent no matter which method is used to check for linear dependence. Since each group of n hidden-layer units is associated with a single linearly dependent pattern set, increasing ϵ_p decreases the number of hidden-layer neurons.

Unfortunately when the number of hidden-layer neurons are decreased by including more patterns in the same group, the overall error of the network tends to increase. One way to reduce this error is to recompute the weight matrix g after the complete set of linearly dependent patterns has been found. The original value of g is computed just as before, using an $m \times m$ section of U. The set of linearly dependent patterns is also computed as before, but then g is recomputed using the complete set of linearly dependent patterns. This is easily done with the block conjugate residual algorithm [16] since it does not require that the u matrix be square. An example of decreasing the number of hidden-layer neurons while keeping the error low with recomputed weights is given in Section IV-B. Further discussion and implementation concerns regarding DI can be found in [16].

D. Further Comments

The algorithm's solution is based on solving a succession of systems of linear equations. Krylov subspace methods, like the block conjugate residual algorithm, see [4] and [16], are recommended since they are iterative methods for solving linear equations, working toward minimizing a quadratic error function instead of attempting to solve the problem exactly in a single complicated step. These methods are guaranteed to converge in a number of steps equal to the order of the system, assuming that the problem is sufficiently well conditioned [12]. The speed of these algorithms is a key to the overall speed of DI.

[1]The bound of p assumes that one input to the network is constant, if the neural thresholds are handled differently then the bound is $p - 1$.

[2]The notation ceiling(x) means the smallest integer greater than or equal to x.

The DI algorithm can not only be used to construct continuous sigmoidal networks but can also be used to construct networks that use discontinuous switching functions. The discontinuities of these functions prevent them from being used with standard gradient-based training.

DI shares some common characteristics with Hecht-Nielsen's counterpropagation networks [5], [7], [8] which are known to converge quickly. Both approaches can be used for continuous function approximation, and both use the hidden layer to organize and classify patterns. DI is primarily distinguished from counterpropagation in that it is a construction algorithm. The architecture of a counterpropagation network is chosen by the designer as well as the supervised and unsupervised learning parameters. Counterpropagation networks perform competitive learning with the hidden-layer neurons, thus their operation often involves "winner-take-all" or similar rules. The networks constructed by DI have a standard feedforward architecture that allows them to be used with tools developed for such networks such as feedforward iterative training techniques and pruning methods. Maintaining the structure of the standard BP-trained feedforward network was an important goal in the development of DI.

DI is, as presented, a batch training process, which might make it inappropriate for certain on-line training tasks. DI, however, is very useful for creating an initial network that may be further trained on-line using some sort of gradient descent (like BP) that responds to each new training pattern as it is presented to the network. The usefulness of combining DI construction with further training from BP is illustrated in Section IV-C.

IV. Examples

Programs are written in C to perform BP and DI and are run on Sun SPARCstation 2 workstations. Section IV-A shows a comparison between networks constructed with DI to networks trained with BP. Section IV-B shows how the number of hidden-layer neurons may be reduced by increasing the DI tolerance value ϵ_p. The overall error of the network is kept down, while the tolerance is increased, by recomputing the weights after each linearly dependent set of training patterns is found. Section IV-C shows how DI may be successfully used to construct a network architecture and set of initial conditions for further training by BP.

A. Function Approximation Examples

DI has been tested on several multi-input/multi-output functions with the number of training patterns ranging from 20 to 2000. The speed and utility of the algorithm is demonstrated by comparing the results obtained from BP training from random initial conditions.

Three function approximation examples are covered in this section. The first is a sine wave

$$d = \sin(u_1)$$
$$u_1 \in [0,\, 4\pi]. \tag{19}$$

TABLE II
Backpropagation Learning Parameters and Stopping Criteria

	η	β	Stopping criterion
Function (19)	0.02 – 0.002	0.1	Square error < 1.0
Function (20)	0.005	0.1	Square error < 0.8
Function (21)	0.005	0.1	1 million iterations

TABLE III
Comparative Results of Network Training Methods

	Training Method	Network Architecture	Square Error (Training)	Square Error (Test Set)	Time to Solution (seconds)
Function (19) 20 patterns	DI	2 – 9 – 1	0.2923	1.8023	0.0167
	BP	2 – 10 – 1	0.9993	3.8570	274.8
Function (20) 200 patterns	DI	3 – 103 – 2	0.3547	1.5508	15.14
	BP	3 – 103 – 2	0.7902	2.1196	4429.25
Function (21) 2000 patterns	DI	4 – 286 – 1	0.4495	0.1221	291.13
	BP	4 – 50 – 1	1.2661	0.2475	3026.93

The second is a two input–two output function given by

$$\begin{aligned} d_1 &= \tfrac{1}{6}\,[5\,\sin(u_1) + \cos(u_2)] \\ d_2 &= \tfrac{1}{5}\,[3\,\cos(u_1) + 2\,\sin(u_2)] \\ u_1,\, u_2 &\in [0,\, 2\pi]. \end{aligned} \tag{20}$$

The third is a three-input/one-output function

$$\begin{aligned} d &= \tfrac{1}{10}\,[e^{u_1} + u_2 u_3\,\cos(u_1 u_2) + u_1 u_3] \\ u_1 &\in [0,\, 1] \\ u_2,\, u_3 &\in [-2,\, 2]. \end{aligned} \tag{21}$$

The activation function for the networks created by BP and DI for all three functions is $\phi(x) = \tanh(x)$. Initial weights for the networks trained with BP were uniformly distributed random values between -0.1 and 0.1. The learning parameters and stopping criteria used in the BP training were obtained through trial and error; a variety of values were tried and the results that yielded the smallest errors in the shortest amount of time are reported here. The stopping criterion for each function was chosen so that the BP cost function was near a local minima and that the resulting error was of the same order of magnitude as that obtained by DI (the use of BP to improve the error obtained from DI is shown in Section IV-C). The parameters used to train each of the given functions with BP are summarized in Table II. The training results are given in Table III.

Twenty training patterns were generated, evenly spaced from $u_1 = 0$ to $u_1 = 4\pi$, to train the sine wave of function (19). DI reached a solution fairly quickly and produced a final square error of 0.2923 in 0.0167 s, while deriving a network architecture of 2–9–1.

BP had significant difficulties with learning the sine wave, being particularly susceptible to local minima in the error space. Many methods were tried to overcome the problem, including varying the number of hidden-layer neurons. The problem was finally overcome by using a network architecture of 2–10–1 and setting the BP learning rate $\eta = 0.02$, and the momentum term $\beta = 0.1$. These learning parameters caused the solution to vary wildly, but it was stopped as soon as it found one set of weights that "beat" the local minimum. These weights were then used as the initial conditions with a slower

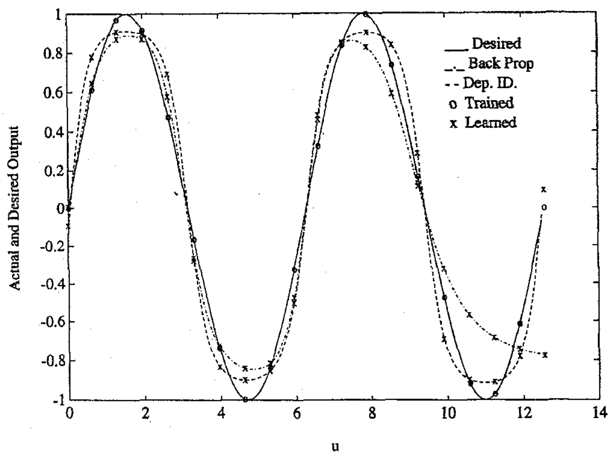


Fig. 3. Testing results of sine wave approximation.

learning rate of $\eta = 0.002$, $\beta = 0.1$ and BP iterations were performed until the final square error was less than 1.0. Table III shows the actual CPU time required to construct and/or train the networks. Note that DI was over 1600 times faster than BP for computing a slightly more accurate solution.

The process of determining BP learning parameters η and β, the number of hidden-layer neurons, and the stopping requirement for BP can be a tedious trial and error procedure. The time required to actually determine an appropriate set of parameters is not included as part of the solution time reported for BP. The times in Table III illustrate that even when the learning rates and network architecture are known, the systems of linear equations used by DI are computed faster than the gradient descent performed by BP. The most important contribution of DI, however, is that the search for the appropriate parameters is eliminated and an appropriate network architecture is an output of the algorithm, not an input.

The two neural-network approximations to function (19) were tested by choosing 126 values of u_1 evenly spaced between zero and 4π. The results are shown in Fig. 3. The 20 training patterns are marked with "o"s on the graph, and the corresponding learned responses to these patterns are marked with "x"s. Table III shows that DI computed a better approximation for the 126 test patterns as well as the training set of 20 patterns while requiring much less training time.

A training set for function (20) is created by generating 200 uniformly distributed random values of u_1 and u_2 and calculating the associated values of d_1 and d_2. The training set for function (21) contains 2000 patterns. DI derived a lower error solution almost 300 times faster than BP for function (20) and 10 times faster for function (21), despite the fact that DI must deal with very large matrices to construct the network for function (21). The number of hidden-layer neurons used by BP was kept low while approximating function (21) due to an excessively long training time that did not yield a corresponding decrease in error.

Fig. 4(a) and (b) shows the results of testing the approximations of function (20) with u_1 and u_2 set to parameterized functions of t, which takes on 100 evenly spaced values within the input range. Fig. 4(c) and (d) show, the results of testing the approximations of function (21). Table III shows that DI performs better on the test sets as well as the training sets.

DI has been compared to BP because BP is a well-known algorithm and many neural-network researchers have experience training feedforward networks with it. The results are intended to demonstrate the utility of DI by contrasting its training times with a training method that the reader knows to be slow and sometimes cumbersome. The results demonstrate that DI is, at the very least, a fast, effective method of developing the initial network architecture and weights from the given training data. Faster methods than BP for training neural-network function approximators are of course known to exist. One of these faster methods is Hecht-Nielsen's counterpropagation networks [7]. Counterpropagation networks actually have a different network architecture than the kind discussed in this paper, and DI is a construction technique while counterpropagation is a training method like BP, but because counterpropagation networks are known to converge quickly, DI was compared to them anyway. Forward-only counterpropagation based on an algorithm described in [5] was implemented on the same computing platform used to test DI and BP. The output of the counterpropagation networks was formed using a linear interpolation of the hidden-layer units in a neighborhood of the winning unit, rather than using strict winner-take-all. In all of the examples above, DI constructed a feedforward network that achieved less error in less time than could be produced using counterpropagation. Because counterpropagation networks are a kind of adaptive lookup table, the error produced by a counterpropagation network can be made very small as the number of hidden neurons approaches the number of training patterns. In this case, however, the training can take a considerably longer time. When counterpropagation was constrained to use the same number of hidden-layer neurons as DI, DI was able to construct a better network approximator in 5–20% of the time required by counterpropagation. Empirical evidence from these examples suggests that DI requires fewer neurons to produce a network with similar error, and it produces networks in less time than that needed by counterpropagation training. Furthermore DI is a construction (rather than training) technique, and it uses the standard feedforward network architecture that may be further modified by established gradient training and pruning techniques.

B. Increased Tolerance and Recomputed Weights

Hardware and/or memory restrictions may limit the number of hidden layers or hidden-layer neurons that may be used in a particular neural network. Section III-C mentions that it is possible to decrease the number of hidden neurons created by the DI algorithm by increasing the tolerance, ϵ_p, used to check whether or not a pattern should be included in a set of "linearly dependent" patterns. Unfortunately as the parameter ϵ_p is increased and the number of hidden-layer neurons decreases, the overall error of the network tends to increase. The increase in network error can be combated by recomputing the weight matrix g (see Algorithm 1) for the entire set of linearly dependent patterns after this set has been

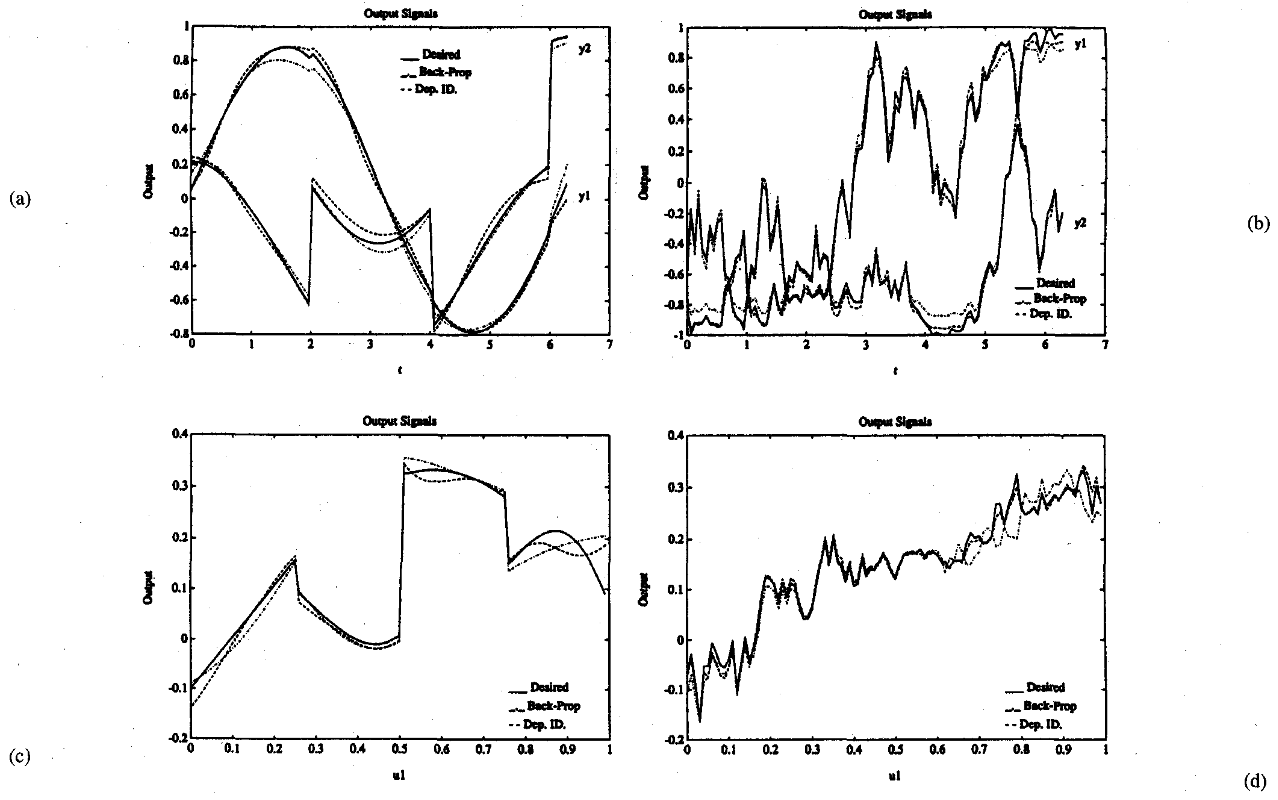


Fig. 4. Testing results of the network approximations.

found. Fig. 5 shows how the number of hidden-layer neurons decreases as the tolerance ϵ_p is increased from 0.05 to 5.00. The training patterns are for the sine wave approximation problem from Section IV-A function (19) and the network is constrained to have three layers[3] ($h_{\max} = 3$). Note that in this example the DI algorithm determines linear dependence on the basis of the quadratic error cost, i.e., it is checking whether or not $ug = v$ instead of $\Phi(ug) = d$. This allows the error tolerance to be increased past $\epsilon_p = 2$ up to $\epsilon_p = 5$. In Section IV-A the dependence judgments were based on the nonlinear, not the quadratic check. Fig. 5 shows that the number of hidden-layer neurons can be successfully decreased with and without the recomputation of the weights. The reason the number of hidden-layer neurons is different when the weights are recomputed is that a three-layer network is being constructed. The inputs produce a set number of neurons in the first hidden layer dependent on ϵ_p only, not on whether the weights are recomputed. The recomputation of weights produces different hidden-layer activations (outputs) than those produced when the weights are not recomputed. Thus the second hidden layer may have a different number of neurons when the weights are recomputed since the inputs to the second hidden layer are different.

Fig. 6 shows how the overall error of the network varies as the tolerance is increased. Network errors are calculated with (2). The figure shows how recomputing the weight matrix helps to keep the overall error of the network from blowing up. It also shows that there is no simple relationship between the overall error and the tolerance ϵ_p. For example, when ϵ_p increases from three to four, the overall error when the weights are not recomputed actually decreases from about 9.2 to 3.2. The trend, however, is that the recomputation keeps the error down. The average error for the 14 different tolerances used in the figure is 3.47 when the weights are not recomputed and 1.25 when they are. It is possible that an error of 1.25 may be higher than the degree of approximation accuracy required for a particular problem. The next section shows how DI can

[3]The networks in Section IV-A have two layers.

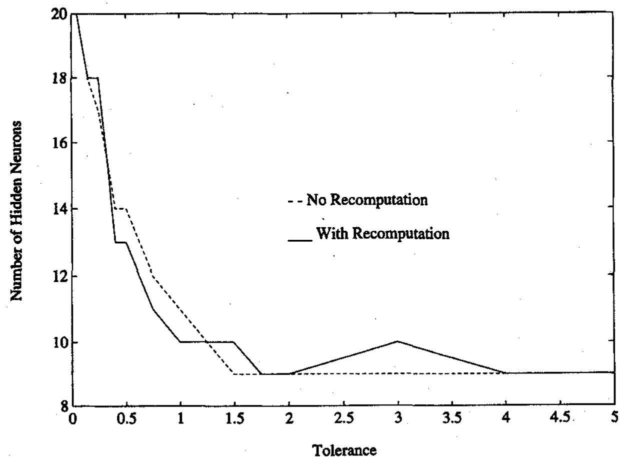


Fig. 5. Decrease of hidden-layer neurons due to increased tolerance ϵ_p.

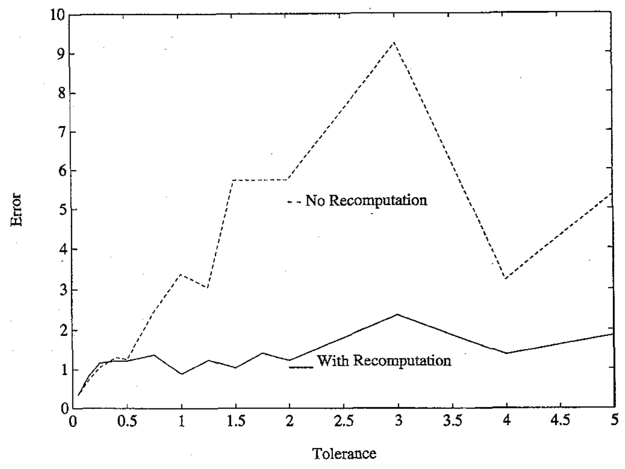


Fig. 6. Overall network error versus increased tolerance ϵ_p.

TABLE IV
RESULTS OF PARITY CHECKER TRAINING

Training Method	Correctly Classified Patterns	% Error
BP with initial conditions from DI	62	3.12%
Dependence Identification	47	26.56%
BP with random initial conditions	30	53.12%

be used to identify an appropriate network architecture and set of initial conditions that will allow BP to quickly decrease the overall network error.

C. DI for BP Initial Conditions

BP is typically sensitive to initial conditions and there are some problems that are inherently very difficult for it to solve due to a large number of local minima and relatively flat sections of the cost function. One of these difficult problems is the parity checker [23]. This example is presented to illustrate how DI and iterative gradient techniques like BP can compliment each other and cooperate to solve a problem that is difficult for either method to solve independently.

A parity checker receives a number of Boolean inputs. In this example, the Boolean values are 1 and -1. The parity checker has one Boolean output that should be 1 when there are an odd number of 1's at the input and -1 when there are an even number of 1's at the input. A six-input parity checker is used in this problem. The number of network inputs is seven because one input is constant and is used to create biases. The number of training patterns is 64, which includes all possible combinations of six Boolean inputs ($2^6 = 64$). DI is used to find an appropriate two-layer network architecture of 7-7-1. Networks are then trained using BP with random initial conditions and with the initial conditions obtained from DI. Table IV shows the results. Patterns are considered to be identified correctly if the network has the proper sign, i.e., a desired output of 1 is classified correctly if the network output is positive and a desired output of -1 is classified correctly if the network output is negative. The entry for "% Error" is calculated as the number of missed patterns divided by 64.

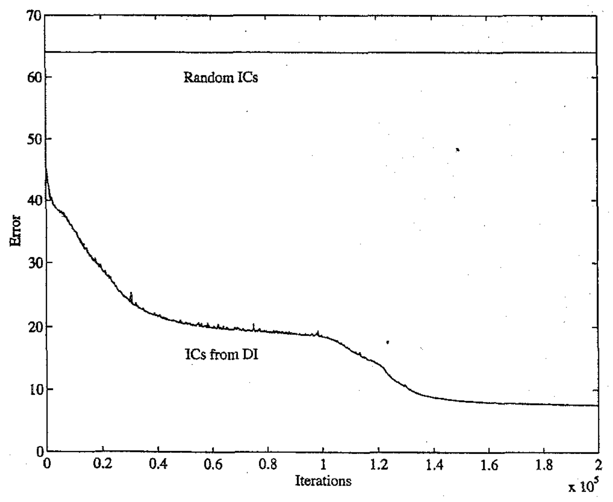


Fig. 7. Network error versus BP iterations for the parity checker problem.

The table shows that, by itself, DI does not produce a very successful solution to the problem. The initial conditions it produces, however, are quite good for improving the network with BP, bringing the error down from about 27% to just over 3%. When random initial conditions are used, BP is completely unsuccessful in learning the problem, misclassifying more than half of the input space.

Fig. 7 shows how the error decreases with the iterations of BP. When initial conditions from DI are used, the error decreases from about 45 to level off around eight. Random initial conditions start out with a high error, and each iteration of BP causes an extremely small change in the overall error. The error curve for random initial conditions (IC's) looks like a straight line with error 64 on the graph, but a magnification would show that the error is slowly going up and down by very slight amounts. The random initial conditions occur in an area of the cost function with a very small slope and probably near a local minimum since the curve goes up and down, instead of steadily down. The graph shows 200 000 iterations. The actual training was continued for 1 000 000 iterations with no change in the behavior of the curve with random initial conditions.

V. CONCLUSIONS

A new method of constructing feedforward multilayer neural networks has been presented, and examples show that it works well for creating neural-network approximations of continuous functions as well as providing a starting point for further BP training. The data from Section IV-A suggest that DI has good generalization properties, however, no formal analysis of this was done and this is a topic for future research.

DI is a faster and more systematic method of developing initial network architectures than trial and error or gradient-based pruning techniques. It is not intended to replace iterative training or pruning methods, as these algorithms can be applied to further refine networks that have been generated with DI. Network size and construction time are conserved since DI builds a small network up instead of whittling a huge network down, and because it relies on the ability to solve fast linear equations instead of ponderous gradient descents. The systems of linear equations are actually a product of the single-layer neural-network training method of quadratic optimization. It would be possible to implement DI with some other single-layer training algorithm, though quadratic optimization is recommended because of its speed compared to techniques involving nonlinear gradient descent.

DI relaxes the constraint that neural activation functions be continuously differentiable, and it determines the number of hidden-layer neurons and layers as part of its operation. DI does not require trial and error with learning rates like BP does. There may well be situations with specific applications where DI indicates a number of hidden-layer neurons that can not be physically implemented (due to memory or hardware constraints). The number of hidden-layer units can be decreased by increasing the tolerance ϵ_p in Algorithm 1. The number of layers can also be limited with the parameter $h_{\max}$. The speed of DI makes it appropriate for creating neural-network architectures and initial weight values to be used in real applications.

APPENDIX

A. Error Cost Function Examples

In this Appendix, three examples are presented that explore and illustrate the relationship between the original single-layer nonlinear minimization problem and the transformed quadratic optimization problem. It is possible to obtain good solutions to the original problem even when quadratic optimization provides significant errors. This is accomplished by working in the "flat" regions of the sigmoid.

Consider a single-layer two-input/one-output neural network with $\phi(x) = \tanh(x)$. The network is capable of realizing functions of the following form exactly

$$y = \tanh(u_1 w_1 + u_2 w_2).$$

The following examples will explore the differences between the nonlinear and quadratic cost functions when the network is trained to approximate a function that can be realized exactly, that has no exact solution, and that has infinite solutions.

B. Unique Zero Error Solution

The network is to be trained to realize the following function

$$d = \tanh(2u_1 + 3u_2)$$

which has an exact solution at $W^* = [2 \quad 3]^T$. The following 10 pattern U and D matrices are created

$$U = \begin{bmatrix} -1.4052 & 0.1485 \\ -2.2648 & 0.8557 \\ 0.8943 & -2.4615 \\ 0.8965 & -0.5829 \\ 2.1735 & -2.1658 \\ -0.5825 & -0.4126 \\ 0.0971 & 0.9339 \\ 1.6548 & 0.4449 \\ -2.3271 & 2.1522 \\ -2.2327 & 1.7308 \end{bmatrix} \quad D = \begin{bmatrix} -0.9825 \\ -0.9613 \\ -1.0000 \\ 0.0443 \\ -0.9732 \\ -0.9838 \\ 0.9950 \\ 0.9998 \\ 0.9471 \\ 0.6212 \end{bmatrix}. \quad (22)$$

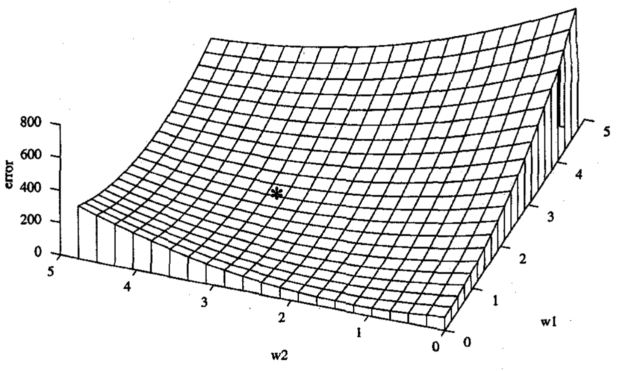


Fig. 8. Quadratic cost function with a unique zero error solution.

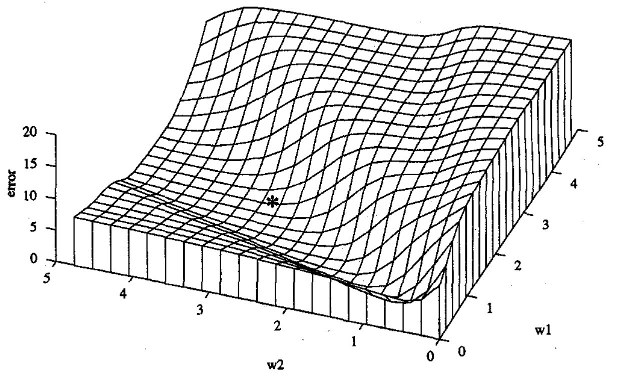


Fig. 9. Nonlinear cost function with a unique zero error solution.

Fig. 8 shows the error described by the quadratic cost function (6). The "*" on the graph at coordinates (2, 3) represents the global minimum that corresponds to $\hat{\epsilon} = 0$ with solution W^*. The previous section demonstrated when $\hat{\epsilon} = 0$ then the solution W^* will also cause $\epsilon = 0$. This is indeed the case as can be seen in Fig. 9, which shows the nonlinear error cost function described by (9). Notice the highly nonlinear nature of the surface shown in Fig. 9. This demonstrates that even with extremely simple networks, the actual nonlinear cost function can be quite complicated.

C. No Exact Solution

In this example, the network can not realize the desired function exactly, but can only approximate it. The desired function is

$$d = 0.9 \tanh(2u_1 + 3u_2).$$

The same U matrix defined in (22) is used and the corresponding new D matrix is created. Fig. 10 shows the quadratic error function. The minimum[4] is given by $\hat{W}^* = [0.75 \quad 1.25]^T$. This minimum is marked by a "Q" on the graph. The minimum for the actual (nonlinear) problem is at $W = [1.25 \quad 2]^T$ and is marked with an "N" on the graph. The nonlinear cost function is shown in Fig. 11 with the two minima marked as above.

[4]The weights in these examples are only computed at intervals of 0.25, thus the actual minimizing weights for the two cost functions may be slightly different than the values given.

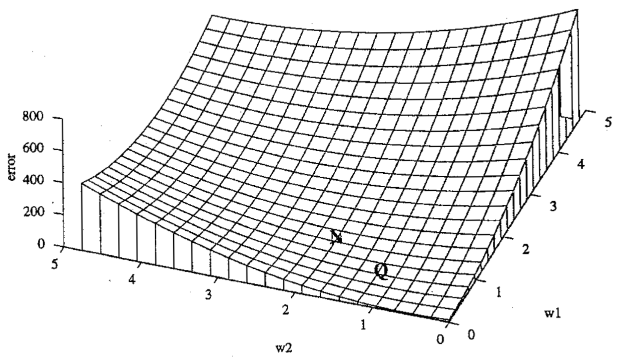


Fig. 10. Quadratic cost function with no zero error solution. Q = quadratic minimum, N = Nonlinear minimum.

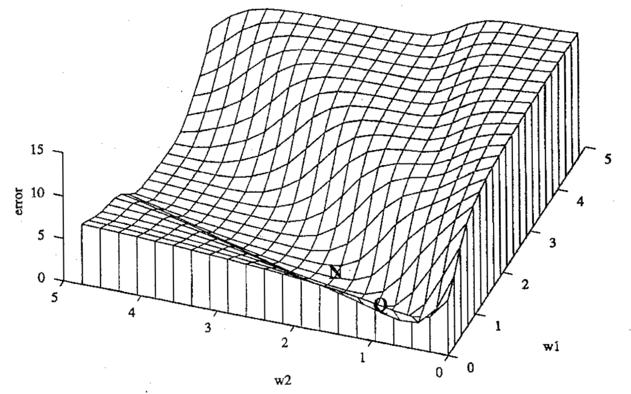


Fig. 11. Nonlinear cost function with no zero error solution. Q = quadratic minimum, N = Nonlinear minimum.

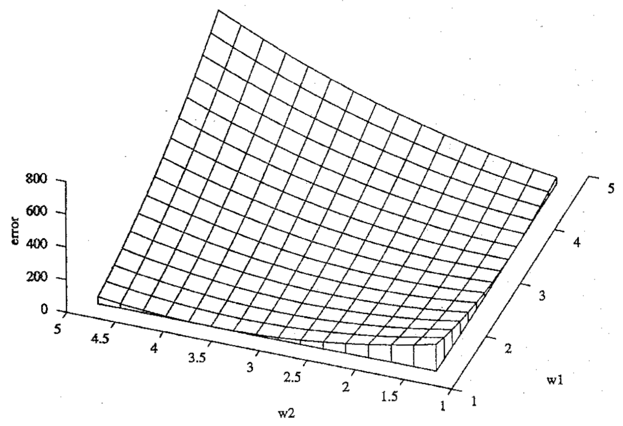


Fig. 12. Quadratic cost function with infinite zero error solutions.

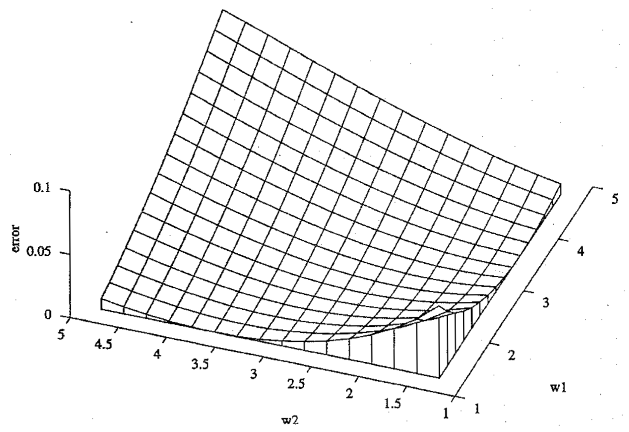


Fig. 13. Nonlinear cost function with infinite zero error solutions.

The derivative values corresponding to the solution of the quadratic function are given by

$$\Phi'(U\hat{W}^*) = \begin{bmatrix} 0.5093 \\ 0.6894 \\ 0.0320 \\ 0.9968 \\ 0.3725 \\ 0.4510 \\ 0.2851 \\ 0.1041 \\ 0.4561 \\ 0.7944 \end{bmatrix}$$

which are not in general near zero, and indicate why the solution to the quadratic problem is not nearer to the solution of the nonlinear problem.

D. Infinite Exact Solution

In this example, there are an infinite number of possible solutions to the problem. The desired function is again

$$d = \tanh(2u_1 + 3u_2) \qquad (23)$$

however, now the U matrix is changed such that both columns are identical. The U and D matrices are now

$$U = \begin{bmatrix} -1.4052 & -1.4052 \\ -2.2648 & -2.2648 \\ 0.8943 & 0.8943 \\ 0.8965 & 0.8965 \\ 2.1735 & 2.1735 \\ -0.5825 & -0.5825 \\ 0.0971 & 0.0971 \\ 1.6548 & 1.6548 \\ -2.3271 & -2.3271 \\ -2.2327 & -2.2327 \end{bmatrix} \quad D = \begin{bmatrix} -1.0000 \\ -1.0000 \\ 0.9997 \\ 0.9997 \\ 1.0000 \\ -0.9941 \\ 0.4506 \\ 1.0000 \\ -1.0000 \\ -1.0000 \end{bmatrix}.$$

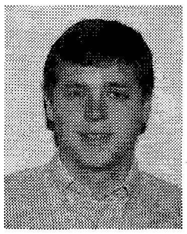
This network has infinite solutions, along the line $w_1 + w_2 = 5$. Figs. 12 and 13 show the quadratic and nonlinear cost functions, respectively. The quadratic cost function has the same line of zero errors as the nonlinear cost.

A possible, and likely, solution to this problem is $\hat{W}^* = [2.5 \quad 2.5]^T$. This is a valid solution to the problem for the particular matrix U, but note that the desired function defined at the beginning of this section has an associated weight matrix of $W = [2 \quad 3]^T$ (23). If the desired function were really $d = \tanh(2u_1 + 3u_2)$ and u_1, and u_2 were allowed to have different values, then we would have found an incorrect solution, not because of some flaw with the training method but because of an unwise choice of U.

The examples above have shown that the method of quadratic optimization will solve the single-layer neural-network training problem exactly if at least one exact solution exists. This is even the case when the nonlinear error cost function is highly irregular or has large flat regions that would hinder gradient descent techniques. An example has shown that when an exact solution to the problem does not exist, then quadratic optimization can come close to the optimal solution, but it will probably not achieve it exactly. The third example demonstrated that care must be taken when choosing the set of training patterns so that the training data accurately reflect the kinds of inputs the trained network will receive. This situation is not unique to quadratic optimization but is present for all training algorithms.

References

[1] E. B. Baum, "On the capabilities of multilayer perceptrons," *J. Complexity,* vol. 4, pp. 193–215, 1988.
[2] D. Chester, "Why two hidden layers are better than one," in *Proc. Int. Joint Conf. Neural Networks,* 1990, pp. 265–268.
[3] G. Cybenko, "Continuous value neural networks with two hidden layers are sufficient," *Math. Contr. Signals Syst.,* vol. 2, pp. 303–314, 1989.
[4] H. C. Elman, "Iterative methods for large sparse nonsymmetric systems of linear equations," Ph.D. dissertation, Comput. Sci. Dep., Yale Univ., New Haven, CT, 1982.
[5] L. Fausett, *Fundamentals of Neural Networks—Architecture, Algorithms, and Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1994.
[6] M. Frean, "The upstart algorithm: A method for constructing and training feedforward neural networks," *Neural Computa.,* vol. 2, pp. 198–209, 1990.
[7] R. Hecht-Nielsen, "Counterpropagation networks," *Appl. Optics,* vol. 26, no. 23, pp. 4979–4984, 1987.
[8] J. Hertz, A. Krogh, and R. G. Palmer, *Introduction to the Theory of Neural Computation.* Reading, MA: Addison-Wesley, 1991.
[9] G. E. Hinton, "Learning distributed representations of concepts," in *Proc. 8th Annu. Conf. Cognitive Sci. Soc.,* 1986.
[10] K. M. Hornik, M. Stinchocombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks,* vol. 2, pp. 359–366, 1989.
[11] L. K. Jones, "Constructive approximations for neural networks by sigmoidal functions," *Proc. IEEE,* vol. 78, no. 10, pp. 1586–1589, 1990.
[12] D. Kincaid and W. Cheney, *Numerical Anal.* Monterey, CA: Brooks/Cole, 1991.
[13] A. H. Kramer and A. Sangiovanni-Vincentelli, "Efficient parallel learning algorithm for neural networks," in *Advances in Neural Information Processing Systems I,* D. S. Touretzky, Ed. San Mateo, CA: Morgan Kaufmann, 1988, pp. 40–48.
[14] M. Marchand, M. Golea, and P. Ruján, "A convergence theorem for sequential learning in two-layer perceptrons," *Europhysics Lett.,* vol. 11, pp. 487–492, 1990.
[15] M. Mézard and J.-P. Nadal, "Learning in feedforward neural networks: The tiling algorithm," *J. Physics,* vol. A, no. 22, pp. 2191–2204, 1989.
[16] J. O. Moody, "A new method for constructing and training multilayer neural networks," Master's thesis, Dep. Electrical Engineering, Univ. Notre Dame, IN, 1993.
[17] N. J. Nilsson, *Learning Machines.* New York: McGraw-Hill, 1965.
[18] F. Rosenblatt, *Principles of Neurodynamics.* New York: Spartan, 1962.
[19] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explanations in the Microstructure of Cognition, vol. 1: Foundation,* D. E. Rumelhart and J. L. McClelland, Eds. Cambridge, MA: MIT Press, 1986, pp. 318–362.
[20] M. A. Sartori and P. J. Antsaklis, "Neural network training via quadratic optimization," in *Proc. ISCAS,* San Diego, CA, May 10–13, 1992, and Tech. Rep. 90-05-01, Dep. Electrical Comput. Eng., Univ. Notre Dame, revised Apr. 1991.
[21] ——, "A simple method to derive bounds on the size and to train multilayer neural networks," *IEEE Trans. Neural Networks,* vol. 2, no. 4, pp. 467–471, July 1991.
[22] R. Scalettar and A. Zee, "Emergence of grandmother memory in feed forward networks: Learning with noise and forgetfulness," in *Connectionist Models and their Implications: Readings from Cognitive Science,* D. Waltz and J. A. Feldman, Eds. Norwood, MA: Ablex, 1988, pp. 309–332.
[23] G. Tesauro and B. Janssens, "Scaling relations in backpropagation learning," *Complex Syst.,* vol. 2, pp. 39–44, 1988.



John O. Moody (S'90) received the B.S. and M.S. degrees in electrical engineering in 1991 and 1993 from the University of Notre Dame, IN, where he is currently pursuing the Ph.D. degree.

He has worked as an engineering intern for the Bendix Engine Controls Division of the Allied Signal Aerospace Company. His research interests include neural network construction and architecture, parallel computing, and autonomous, intelligent control systems.

Mr. Moody is a member of Eta Kappa Nu and Tau Beta Pi.

Panos J. Antsaklis (S'74–M'76–SM'86–F'91) received the Diploma in mechanical and electrical engineering from the National Technical University of Athens (NTUA), Greece, in 1972 and the M.S. and Ph.D. degrees in electrical engineering from Brown University, Providence, RI, in 1974 and 1977, respectively.

He is currently Professor of Electrical Engineering at the University of Notre Dame. He has held faculty positions at Brown University, Rice University, and Imperial College, University of London. During sabbatical leaves, he has been Senior Visiting Scientist at LIDS of MIT in 1987 and at Imperial College in 1992; he also was Visiting Professor at NTUA Greece in 1992. His research interests include multivariable system and control theory, hybrid and discrete event systems, neural networks, and autonomous, intelligent, and learning control systems. He is co-Editor with K. M. Passino of "*An Introduction to Intelligent and Autonomous Control,*" (Kluwer, 1993).

Dr. Antsaklis was the Program Chair of the 30th IEEE CDC in England in 1991 and has been Associate Editor of IEEE TRANSACTIONS ON AUTOMATIC CONTROL and IEEE TRANSACTIONS ON NEURAL NETWORKS. He was the Guest Editor of the 1990 and 1992 special issues on "Neural Networks in Control Systems" and of the 1995 special issue in "Intelligent Learning Control" in the IEEE *Control Systems Magazine*. He has served as the General Chair of the 1993 8th IEEE International Symposium on Intelligent Control in Chicago and of the 1995 34th Conference on Decision and Control in New Orleans. He is an elected member of the IEEE Control Systems Society Board of Governors and Vice President Conferences for 1994–1995.