The Dependence Identification

Neural Network Construction Algorithm

Technical Report of the ISIS Group
at the University of Notre Dame
ISIS-93-005
October, 1993

John O. Moody and Panos J. Antsaklis
Department of Electrical Engineering
University of Notre Dame
Notre Dame, IN 46556

Interdisciplinary Studies of Intelligent Systems

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 and (ii) it constructs an appropriate network to meet the training specifications. The architecture and network weights produced by the algorithm can also be used as initial conditions for further on-line training by backpropagation or a similar iterative gradient descent training algorithm if necessary. In addition to constructing an appropriate network based on training data, the dependence identification algorithm significantly speeds up learning in feedforward multilayer neural networks compared to standard backpropagation.

1 Introduction

The main tool for training multilayer neural networks is gradient descent, as used by the backpropagation (BP) algorithm developed by Rumelhart [17]. 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 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. Backpropagation 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 backpropagation. A method of quickly specifying a network architecture and set of intitial 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 rarely used weights and/or neurons. The major tool for pruning weights and neurons is a method known as "weight decay;" see [6], [7], [11], and [20]. Weight decay causes all of the weights in the neural network to decrease slightly in magnitude at every backpropagation 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 which 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 descent. Hardware or time restrictions may interfere with the size requirements for the initial network, and gradient descent has its own host of problems. Pruning methods still require the designer to guess at the number of layers required for the network, though results from [2], [3], [8] and [9] making guessing at the required number of hidden layers a relatively easy task.

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 algorithm presented in section 3. For this reason several methods of constructing boolean networks are outlined below, details can be found in [5], [6], [12], and [13].

Marchand et al. [12] 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 which 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* [16] 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 [5] 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 the secondary neurons, and the process is repeated until the first neuron correctly classifies all of the I/O 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 [13] 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 which 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 dependence identification (DI) algorithm is presented in this paper. Dependence identification 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 which are suitable for further iterative or on-line adaptive training with gradient descent.

The single layer training method of quadratic optimization, a major tool used by the dependence identification algorithm, is explained in the following section. The dependence identification algorithm is then presented in section 3. Section 4 gives examples of neural network construction using dependence identification. Concluding remarks appear in section 5.

2 Quadratic Optimization

Many methods exist [6] for training *single layer* neural networks including steepest descent, least mean squares, and the "perceptron training algorithm" (for single layer classification problems). In [18], 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 dependence identification algorithm described in the next section. A description of the method is given here.

Individual neurons perform the following function:

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

The neuron receives m inputs, $(u_1, \ldots, u_m)$, 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 j^{th} output may be expressed as

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

where $1 \le j \le n$ and w_{ij} is the weight between the i^{th} input and the j^{th} 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

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

and the ij^{th} 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 $\Phi(uW)$:

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

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:

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

where u_i^k is the value of the i^{th} input for the k^{th} training pattern and d_j^k is the j^{th} desired output for the k^{th} training pattern. The following matrices can then be constructed:

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

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

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

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 equation (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:

$$e = \text{ trace } ((D - Y)^T (D - Y)) = \text{ trace } ((D - \Phi(UW))^T (D - \Phi(UW))) \tag{3}$$

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

$$\min_W \text{ trace } ((D - \Phi(UW))^T (D - \Phi(UW))) \tag{4}$$

In order to deal with the nonlinearity of equation (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 which will evaluate to 1 and an infinite number of inputs which will evaluate to 0. V can still be defined by assigning some positive number to v_j^k whenever d_j^k is 1 and assigning some negative number to v_j^k whenever d_j^k is 0. The numbers assigned may be random or some pre-chosen constants. With the change of variables in effect, the single layer neural network training problem is transformed to

$$\min_W \text{trace}\ ((V - UW)^T (V - UW)) \tag{6}$$

which is equivalent to solving

$$U^T UW = U^T V \tag{7}$$

for W if $U^T U$ 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 (4) 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.

2.1 Error Analysis

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

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

where

$$F(W) = \text{trace}\ (D - \Phi(UW))^T (D - \Phi(UW)) \tag{9}$$

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

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

Note that the scalar error measure e in equations (2) and (3) is $e = \text{trace}\ \epsilon^T \epsilon$. The jk^{th} element of ϵ is the error of the j^{th} output neuron to the k^{th} pattern:

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

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

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

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

where

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

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

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

The relation [18] 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:

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

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

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

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

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

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

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

It is now clear that for ϵ_j^k to be small $\hat{\epsilon}_j^k$ should be small and/or $\phi'(\sum_{i=1}^{m} u_i^k w_{ij})$ should be small. That is, good solutions to the original nonlinear problem are achieved when the weights w_{ij} are such that $\phi'(\sum_{i=1}^{m} u_i^k w_{ij}) \approx 0$, 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 equation (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, equation (16), we want

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

where w_{ij}^* and w_{ij} are the ij^{th} 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:

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

Let α_j^k equal the absolute value of the ratio of the slopes for the j^{th} output neuron and the k^{th} training pattern:

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

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

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

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.

3 Dependence Identification

This section introduces the multilayer neural network construction method dependence identification. An example of network construction is given as an introduction in section 3.1 before the algorithm is formally described in section 3.2. Sections 3.3 and 3.4 address specific issues of the algorithm.

3.1 A Network Construction Example

A pseudo code description of the dependence identification algorithm is given in the next subsection. A simple example is given here in order 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 which 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 below in table 1.

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

Table 1: The XOR function

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

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

Because the I/O space is boolean, it makes sense to define the neural activation function ϕ as the step function which also has output values of 0 and 1. We now need to define the V matrix such that $\Phi(V) = D$. As mentioned in section 2, the step function is not bijective, but we can still assign values to V which 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 iff

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

but U is rank 3 and $[U \ V]$ 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

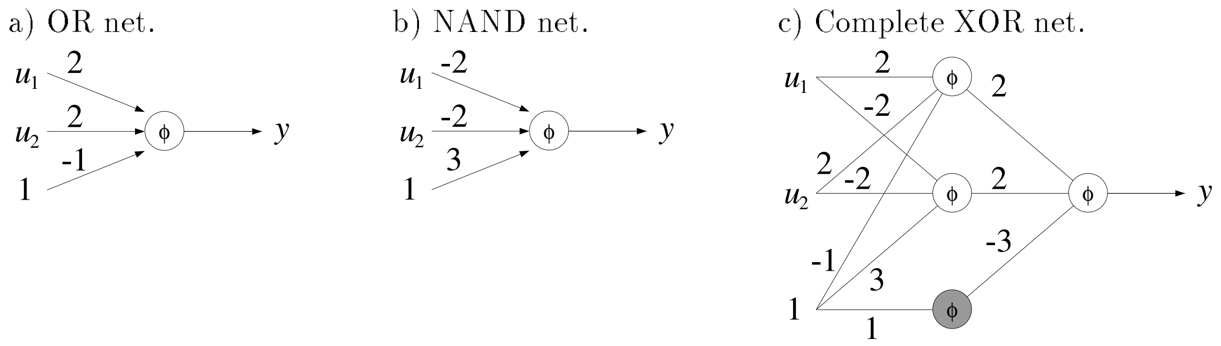

Figure 1: Components of the XOR neural network.

separable which is a well known result. If the XOR function is to be realized with the given non-decreasing activation function, then the network must have a minimum of two layers.

The rank condition (18) dictates the *entire* pattern set can not 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 rank $(U_{1:3}) =$ 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 Figure 1a.

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 equation (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 Figure 1b.

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 which has an output that 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 rank (H) = 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 which realizes the XOR problem is shown in Figure 1c.

3.2 The Algorithm

The XOR network in the previous section was constructed using the dependence identification (DI) algorithm which 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 2. 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 2. 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 equation (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 equations (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 which are correctly matched by a single hidden layer neuron are linearly dependent, which is the reason for the name dependence identification. 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.

The algorithm is presented formally in Algorithm 3.1. 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.

Algorithm 3.1 (Dependence Identification).

```
input U ∈ ℝ^{p×m}, D ∈ ℝ^{p×n}, φ : ℝ → ℝ, ε_N, ε_p, h_max.
Create V such that Φ(V) = D
Set h = 1
repeat
   Solve min_{W_h} trace ((U W_h − V)^T (U W_h − V)), W_h ∈ ℝ^{m×n}.
   Compute error:
   e = trace ((Φ(U W_h) − D)^T (Φ(U W_h) − D))
   if (e > ε_N) and (h < h_max) then
      Set G = empty matrix, l = 0
      Set U_unmarked = U, V_unmarked = V
      repeat
         Choose u ⊂ U_unmarked, v ⊂ V_unmarked where u ∈ ℝ^{m×m},
           v ∈ ℝ^{m×n}
         if less than m patterns are unmarked then augment
           u and v with previously marked patterns.
         Solve:
         min_g trace ((ug − v)^T (ug − v)), g ∈ ℝ^{m×n}.
         Add the solution g as new column(s) of G.
         l ← l + n
         Mark all patterns which do not satisfy U_unmarked g =
           V_unmarked within the tolerance ε_p.
      until all patterns have been marked.
      Find the hidden layer outputs:
      H = Φ(UG) ∈ ℝ^{p×l}.
      Add a zero column to G with a single nonzero element
        corresponding to the constant input.
      W_h ← G
      U ← H
      m ← l
      h ← h + 1
until (e ≤ ε_N) or (h = h_max)
output W_i for i = 1, ..., h and the error e
```

3.3 Hidden Layer Neurons

The dependence identification algorithm constructs a network which solves the *single layer problem* within the desired accuracy, if such a 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], [15], and [19] for constructive proofs.

The bound on the number of hidden layer neurons created by dependence identification 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$(\frac{p}{m})$ linearly dependent groups[2] in order 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$(\frac{p}{m})$ 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 which has a constant output is then added to the hidden layer in order to help form constant offsets and/or thresholds for the output layer. Thus the dependence identification algorithm has an upper bound of ceiling$(\frac{p}{m})n + 1$ hidden layer neurons for a two network and assuming the $m \times m$ portions of U are full rank. The dependence identification algorithm assumes that one input is a constant used to form thresholds for the first layer's neurons. Thus for a SISO network $m = 2$ and $n = 1$ and the bound on the number of hidden layer neurons created by dependence identification is approximately one half the bound of p.

There may be situations where dependence identification 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 3.1, 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 associate 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.

[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.

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 [14] 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 4.2. Further discussion and implementation concerns regarding dependence identification can be found in [14].

3.4 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 [14], 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 [10]. The speed of these algorithms is a key to the overall speed of dependence identification.

The dependence identification 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 the these functions prevent them from being used with standard gradient descent training.

Dependence identification is, as presented, a batch training process, which might make it inappropriate for certain on-line training tasks. However dependence identification is very useful for creating an initial network that may be further trained on-line using some sort of gradient descent (like backpropagation) that responds to each new training pattern as it is presented to the network. The usefulness of combining dependence identification construction with further training from backpropagation is illustrated in section 4.3.

4 Examples

Programs are written in C to perform backpropagation (BP) and dependence identification (DI) and are run on Sun SPARCstation 2 workstations. Section 4.1 shows a comparison between networks constructed with dependence identification to networks trained through trial and error and backpropagation. Section 4.2 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 4.3 shows how dependence identification may be successfully used to construct a network architecture and set of initial conditions for further training by backpropagation.

4.1 Function Approximation Examples

Dependence identification has been tested on several multi-input/multi-output functions with the number of training patterns ranging from 4 to 2000. The functions were also trained using backpropagation with random initial conditions.

Three function approximation examples are covered in this section. The first is a 1 input/1 output sine wave:

$$\begin{aligned} d &= \sin(u_1) \\ u_1 &\in [0, 4\pi] \end{aligned} \tag{19}$$

The second is a 2 input/2 output function given by

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

The third is a 3 input/1 output function:

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

The activation function for the networks created by backpropagation (BP) and dependence identification (DI) for all three functions is $\phi(x) = \tanh(x)$. The training results are summarized in table 2.

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

Backpropagation 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 $\alpha = 0.02$, 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

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

Table 2: Comparative results of network training methods.

local minimum. These weights were then used as the initial conditions with a slower learning rate of $\alpha = 0.002, \beta = 0.1$ and BP iterations were performed until the final square error was less than 1.0. Table 2 shows that dependence identification was over 1600 times faster than backpropagation and found a better solution.

The process of determining BP learning parameters α and β, the number of hidden layer neurons, and the stopping requirement for backpropagation is 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 for backpropagation. The times in table 2 illustrate that even when the learning rates and network architecture are known, the systems of linear equations used by dependence identification are computed faster than the gradient descent performed by backpropagation. However the most important contribution of dependence identification 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 0 and 4π. The results are shown in figure 2. The twenty training patterns are marked with 'o's on the graph, and the corresponding learned responses to these patterns are marked with 'x's. Table 2 shows that *DI did better at matching all of the 126 test patterns* as well as performing better on the training set of twenty patterns.

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. The learning parameters for back-propagation are $\eta = 0.005$ and $\alpha = 0.1$, where η is the learning step size along the cost gradient and α is the momentum factor. The BP iterations are halted when the overall square error is less than 0.8 for function (20) and after 1 million BP iterations for function (21). The training times given in table 2 are the actual CPU times used to calculate the solutions. Note that dependence identification derived a lower error solution almost 300 times faster than back-propagation for function (20) and 10 times faster for function (21), despite the fact that DI must deal with very large matrices in order 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

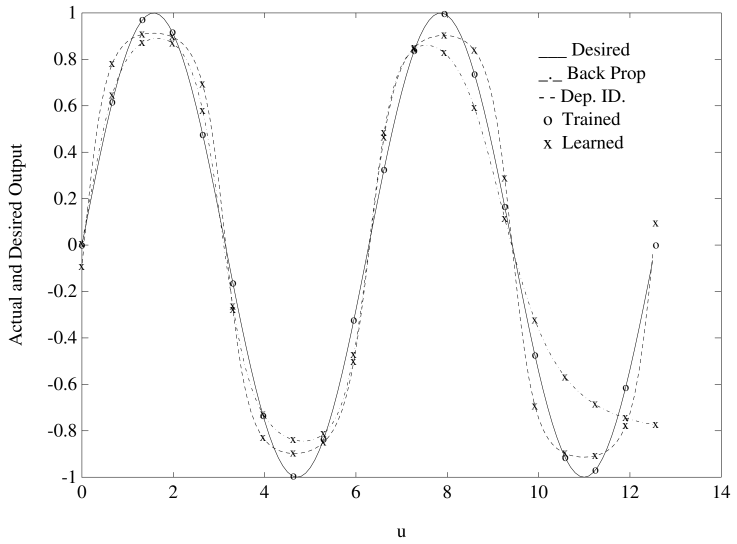

Figure 2: Testing results of sine wave approximation.

the excessively long training times required.

Figures 3a and 3b show the results of testing the approximations of function (20) with u_1 and u_2 set to parametrized functions of t, which takes on 100 evenly spaced values within the input range. Figures 3c and 3d show the results of testing the approximations of function (21). Table 2 shows that dependence identification performs better on the test sets as well as the training sets.

4.2 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 3.3 mentions that it is possible to decrease the number of hidden neurons created by the dependence identification 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 3.1) for the entire set of linearly dependent patterns after this set has been found. Figure 4 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 4.1 function (19) and the network is constrained to have three layers[3] ($h_{\max} = 3$). Note that in this example the dependence identification 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

[3]The networks in section 4.1 have two layers

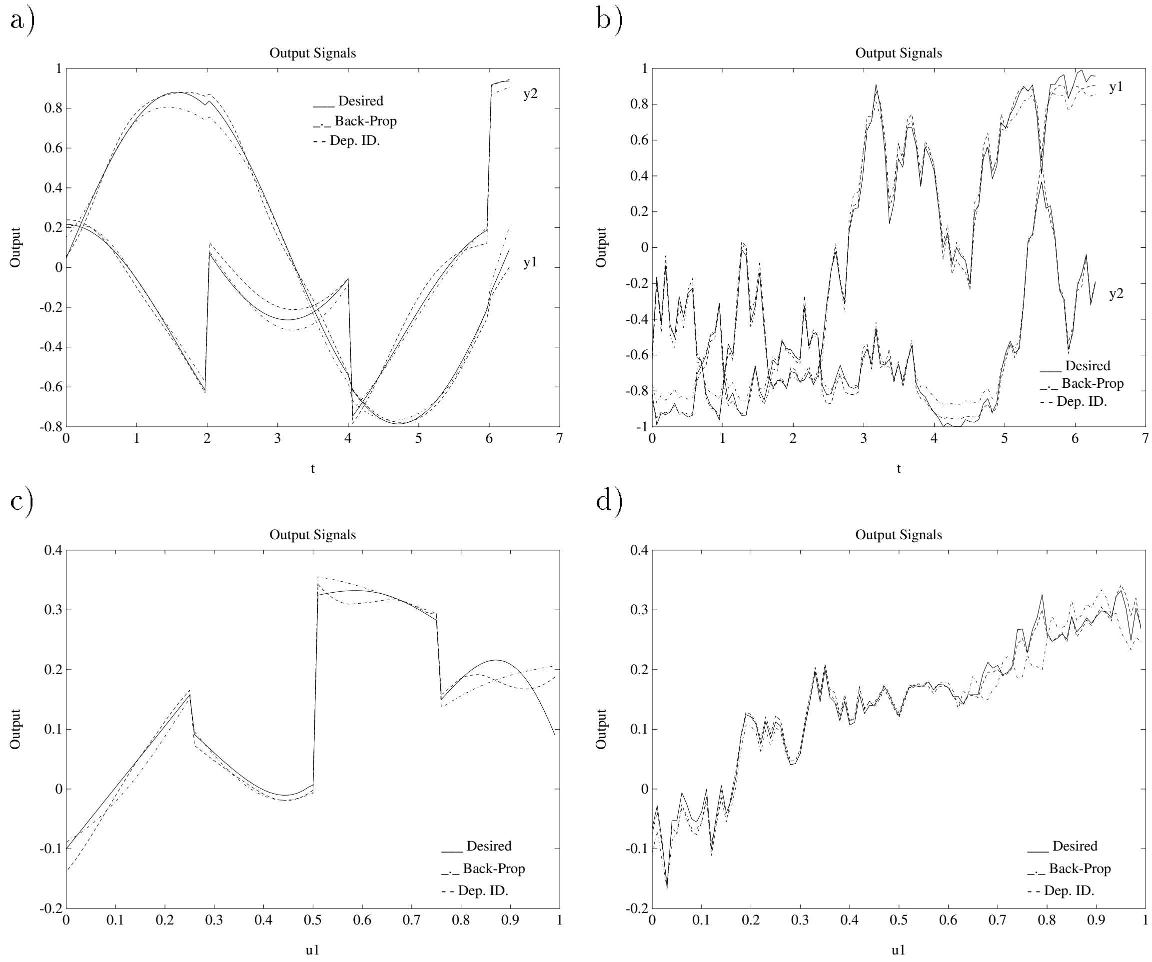

Figure 3: Testing results of the network approximations.

increased past $\epsilon_p = 2$ up to $\epsilon_p = 5$. In section 4.1 the dependence judgements were based on the nonlinear, not the quadratic check. Figure 4 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.

Figure 5 shows how the overall error of the network varies as the tolerance is increased. Network errors are calculated with equation (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 3 to 4, the overall error when the weights are not recomputed actually decreases from about 9.2 to 3.2. However the trend is that the recomputation keeps

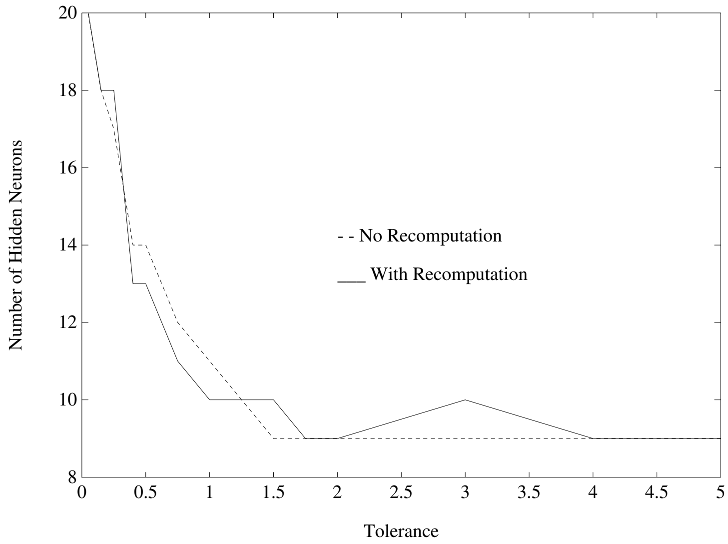

Figure 4: Decrease of hidden layer neurons due to increased tolerance ϵ_p.

the error down. The average error for the fourteen 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 dependence identification can be used to identify an appropriate network architecture and set of initial conditions that will allow backpropagation to quickly decrease the overall network error.

4.3 Dependence Identification for Backpropagation Initial Conditions

Backpropagation is extremely sensitive to initial conditions and there are some problems that are inherently very difficult for backpropagation 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 [21]. 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 which should be 1 when there is 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 ($2^6 = 64$). Dependence identification is used to find an appropriate two layer network architecture of $7 - 7 - 1$. Networks are then trained using backpropagation with random initial conditions and with the initial conditions obtained from dependence identification. Table 3 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"

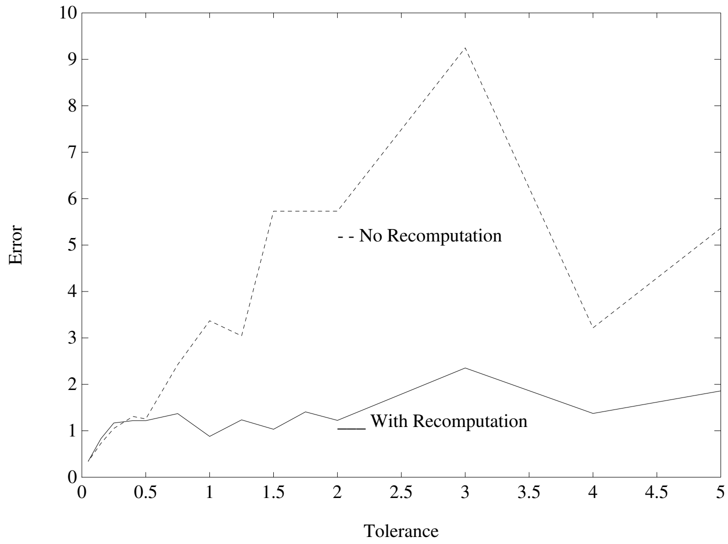

Figure 5: Overall network error vs. increased tolerance ϵ_p.

is calculated as the number of missed patterns divided by 64.

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

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%

Table 3: Results of parity checker training

Figure 6 shows how the error decreases with the iterations of backpropagation. When initial conditions from dependence identification are used, the error decreases from about 45 to level off around 8. Random initial conditions start out with a high error, and each iteration of backpropagation causes an extremely small change in the overall error. The error curve for random initial conditions (ICs) 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 (which required approximately 100 seconds to calculate). The actual training was continued for one million iterations with no change in the behavior of the curve with random initial conditions.

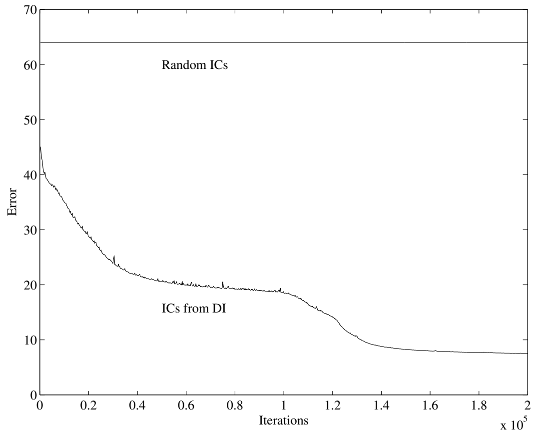

Figure 6: Network error vs. BP iterations for the parity checker problem.

5 Conclusions

A new method of constructing feedforward multi-layer 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 backpropagation training. Dependence identification is a faster and more systematic method of developing initial network architectures than trial and error or pruning techniques used with gradient descent. Network size and construction time are conserved since dependence identification 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 dependence identification with some other single layer training algorithm, though quadratic optimization is recommended because of its speed compared to techniques involving gradient descent.

Dependence identification 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. Dependence identification does not require trial and error with learning rates like backpropagation 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 3.1. 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
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.

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 $W^* = \begin{bmatrix} 2 & 3 \end{bmatrix}^T$. The following ten pattern U and D matrices are created:

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

Figure 7 shows the error described by the quadratic cost function equation (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 figure 8 which shows the nonlinear error cost function described by equation (9). Notice the highly nonlinear nature of the surface shown in figure 8. This demonstrates that even with extremely simple networks, the actual nonlinear cost function can be quite complicated.

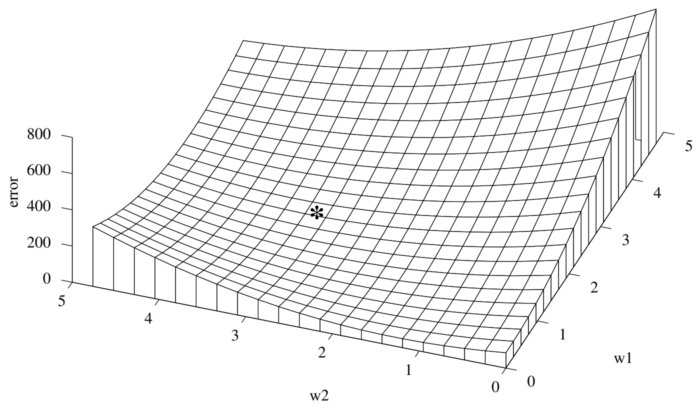

Figure 7: Quadratic cost function with a unique zero error solution.

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 equation (22) is used and the corresponding new D matrix is created. Figure 9 shows the quadratic error function. The minimum[4] is given by $\hat{W}^* = \begin{bmatrix} 0.75 & 1.25 \end{bmatrix}^T$. This minimum is marked by a 'Q' on the graph. The minimum for the actual (nonlinear) problem is at $W = \begin{bmatrix} 1.25 & 2 \end{bmatrix}^T$ and is marked with an 'N' on the graph. The nonlinear cost function is shown in figure 10 with the two minima marked as above.

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

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

[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.

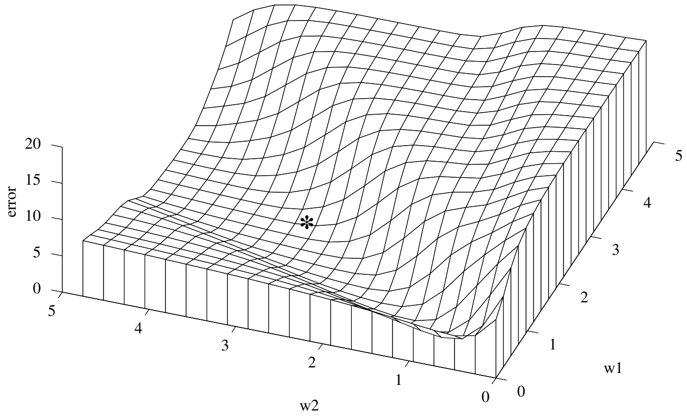

Figure 8: Nonlinear cost function with a unique zero error solution.

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.

Infinite Exact Solution

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

$$d = \tanh(2u_1 + 3u_2) \tag{23}$$

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

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

This network has infinite solutions, along the line $w_1 + w_2 = 5$. Figures 11 and 12 show the quadratic and nonlinear cost functions respectively. The quadratic cost function has the same line of zero errors as the the nonlinear cost.

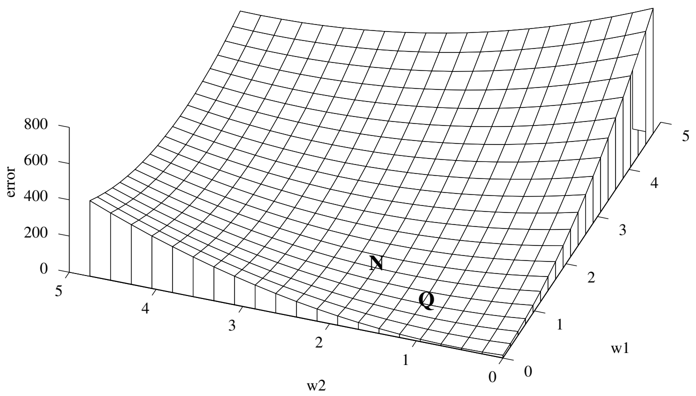

Figure 9: Quadratic cost function with no zero error solution. Q = quadratic minimum, N = Nonlinear minimum.

A possible, and likely, solution to this problem is $\hat{W}^* = \left[\begin{array}{cc} 2.5 & 2.5 \end{array}\right]^T$. 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 $W = \left[\begin{array}{cc} 2 & 3 \end{array}\right]^T$ (equation (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 reflects 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] Baum E. B., "On the Capabilities of Multilayer Perceptrons," *J. Complexity*, vol 4, pp 193–215, 1988.

[2] Chester D. (1990), "Why Two Hidden Layers are Better than One," *Proc Int. Joint Conf. on Neural Networks*, IEEE Publications, pp. 265–268, 1990.

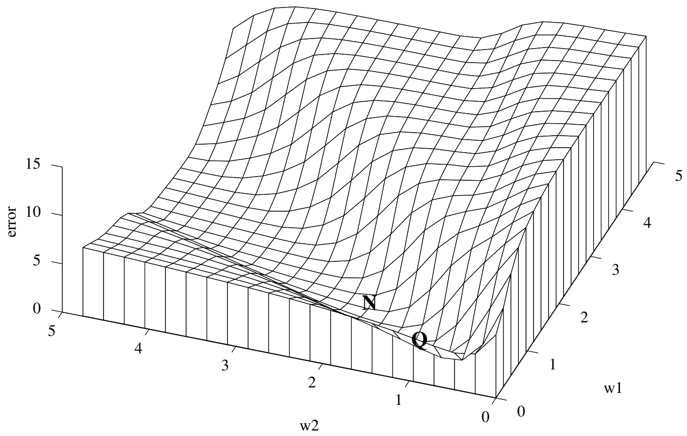

Figure 10: Nonlinear cost function with no zero error solution. Q = quadratic minimum, N = Nonlinear minimum.

[3] Cybenko, G., "Continuous Value Neural Networks with Two Hidden Layers are Sufficient," *Math. Contr. Signals & Systems*, vol 2, pp. 303–314, 1989.

[4] Elman, H. C., "Iterative Methods for Large Sparse Nonsymmetric Systems of Linear Equations," Ph.D. thesis, Computer Science Dept., Yale University, New Haven, CT., 1982.

[5] Frean, M., "The Upstart Algorithm: A Method for Constructing and Training Feedforward Neural Networks," *Neural Computation* 2, pp. 198–209, 1990.

[6] Hertz, J., Krogh, A. and Palmer, R. G., *Introduction to the Theory of Neural Computation,* Addison-Wesley Publishing Company, 1991.

[7] Hinton, G. E., "Learning Distributed Representations of Concepts," *Proceedings of Eighth Annual Conference of the Cognitive Science Society*, Amherst, 1986. Hillsdale: Erlbaum.

[8] Hornik K. M., Stinchocombe M., White H., "Multilayer Feedforward Networks are Universal Approximators," *Neural Networks*, vol. 2, pp. 359–366, 1989.

[9] Jones L. K., "Constructive Approximations for Neural Networks by Sigmoidal Functions," *IEEE Proceedings*, vol 78, no 10, pp 1586–1589, 1990.

[10] Kincaid, D. and Cheney, W., *Numerical Analysis*, Brooks/Cole Publishing Company, 1991.

[11] Kramer, A. H., Sangiovanni-Vincentelli, A., "Efficient Parallel Learning Algorithm for Neural Networks," *Advances in Neural Information Processing Systems I*, ed. D. S. Touretzky, pp. 40–48, Denver, 1988. San Mateo: Morgan Kaufmann.

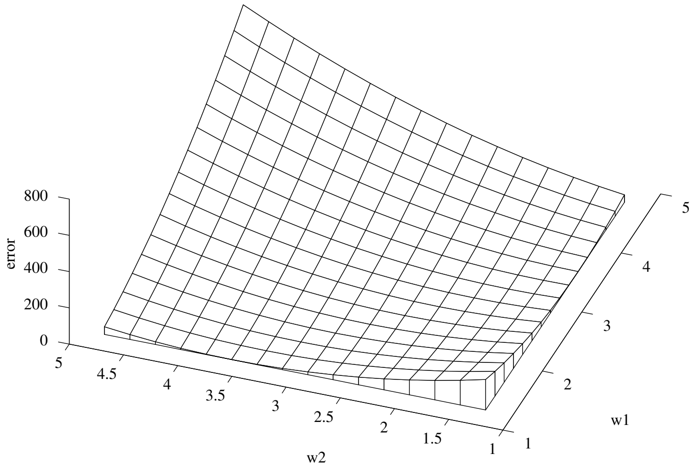

Figure 11: Quadratic cost function with infinite zero error solutions.

[12] Marchand. M., Golea, M., Ruján, P., "A Convergence Theorem for Sequential Learning in Two-Layer Perceptrons," *Europhysics Letters* 11, pp. 487–492, 1990.

[13] Mézard, M., Nadal, J. -P., "Learning in Feedforward Neural Networks: The Tiling Algorithm," *Journal of Physics* A 22, pp. 2191–2204, 1989.

[14] Moody, J. O., "A New Method for Constructing and Training Multilayer Neural Networks," Masters thesis, Dept. of Electrical Engineering, University of Notre Dame, Notre Dame, IN., 1993.

[15] Nilsson N. J., *Learning Machines*, McGraw-Hill, 1965.

[16] Rosenblatt, F., *Principles of Neurodynamics*, Spartan, New York, 1962.

[17] Rumelhart, D. E., Hinton, G. E. and Williams, R. J., "Learning Internal Representations by Error Propagation," in Rumelhart, D. E. and McClelland, J. L., eds. *Parallel Distributed Processing: Explanations in the Microstructure of Cognition, vol 1: Foundation*, pp. 318–362, MIT Press, 1986.

[18] Sartori, M. A. and Antsaklis, P. J., "Neural Network Training via Quadratic Optimization", Proc of ISCAS, San Diego, CA, May 10–13, 1992.

[19] Sartori, M. A. and Antsaklis, P. J., "A Simple Method to Derive Bounds on the Size and to Train Multi-Layer Neural Networks", IEEE Trans. on Neural Networks, Vol 2, No 4, pp. 467–471, July 1991.

[20] Scalettar, R., Zee, A., "Emergence of Grandmother Memory in Feed Forward Networks: Learning with Noise and Forgetfulness," *Connectionist Models and their Implications: Readings from Cognitive Science*, eds. D. Waltz and J. A. Feldman, pp. 309–332. Norwood: Ablex, 1988.

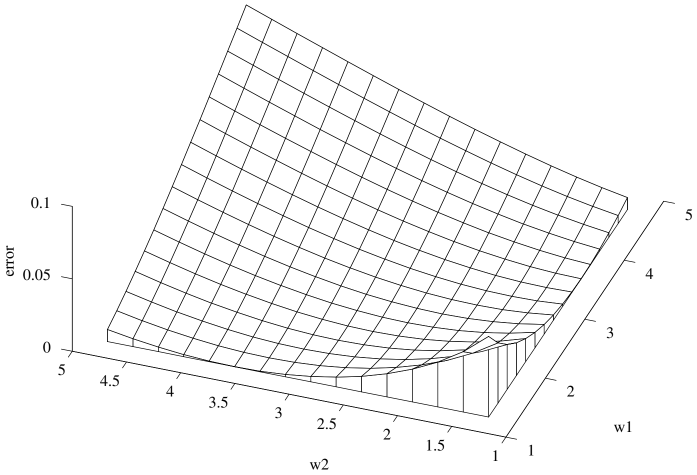

Figure 12: Nonlinear cost function with infinite zero error solutions.

[21] Tesauro, G. and Janssens, B., "Scaling Relations in Backpropagation learning", *Complex Systems*, Vol 2 pp. 39–44, 1988.