# Synthesis of Fault-Tolerant Feedforward Neural Networks Using Minimax Optimization

Dipti Deodhare, M. Vidyasagar, *Fellow, IEEE,* and S. Sathiya Keerthi

***Abstract*—In this paper we examine a technique by which fault tolerance can be embedded into a feedforward network leading to a network tolerant to the loss of a node and its associated weights. The fault tolerance problem for a feedforward network is formulated as a constrained minimax optimization problem. Two different methods are used to solve it. In the first method, the constrained minimax optimization problem is converted to a sequence of unconstrained *least-squares* optimization problems, whose solutions converge to the solution of the original minimax problem. An efficient gradient-based minimization technique, specially tailored for nonlinear least-squares optimization, is then applied to perform the unconstrained minimization at each step of the sequence. Several modifications are made to the basic algorithm to improve its speed of convergence. In the second method a different approach is used to convert the problem to a single unconstrained minimization problem whose solution very nearly equals that of the original minimax problem. Networks synthesized using these methods, though not always fault tolerant, exhibit an acceptable degree of partial fault tolerance.**

***Index Terms*—Fault tolerance, minimax optimization.**

## I. Introduction

DISTRIBUTED computing, of which neural networks are an example, promises to offer new ways of achieving fault tolerance. By virtue of their architecture and processing style neural networks are believed to be inherently fault tolerant. When one unit fails other units may partially take over its functions, so as to achieve *graceful degradation* or *fail-soft* performance. Artificial neural networks are inspired by natural neural networks in the human brain and consist of distributed processing elements with each node contributing to the final output response. The human brain exhibits a remarkable degree of fault tolerance since it continues to function in spite of losing as many as $10^4$ neurons per day. Fault tolerance is therefore a desirable property and is *believed* to be an intrinsic property of (artificial) neural networks. The main reason advanced for this belief is the fact that the storage mechanism is *connectionist* and cutting off a few neurons and their associated interconnections presumably should not affect the performance of the network drastically. Thus, if a node or its weights are lost or damaged, recall is impaired in quality, but the distributed nature of the information storage means that damage has to be extensive before the response of the network degrades badly. The network therefore demonstrates graceful degradation instead of catastrophic failure. However, there is thus far very little *theoretical basis* for such a belief. The ability of the network to exhibit fault tolerance depends on the algorithm used to train it. Most connectionist or neural-network learning systems use some version of the backpropagation algorithm which has been shown to produce networks that are not always fault tolerant [1]. Discussion of formal techniques that produce feedforward neural networks *guaranteed* to be fault tolerant has been limited in the literature. Perhaps this is because of the fact that embedding fault tolerance into the network design/training process is a nontrivial problem.

The backpropagation method was introduced to overcome the problem of structural credit assignment in a multilayer perceptron network (MLPN). This algorithm has become almost synonymous with MLPN's to such an extent that a clear distinction between the architecture and the training algorithm has been lost in many cases. However, this method is only *one particular* way of determining the weights in an MLPN so that it performs a specific task. Moreover, it is not uncommon to see that, when a feedforward neural network is trained to recognize a given set of patterns using the backpropagation method, the network fails to reproduce the specified input–output pairs when one or more neurons is eliminated from the network [1], [3]. An MLPN trained by the backpropagation algorithm can easily be subjected to fault tolerance analysis by artificially forcing randomly selected weights to zero. A similar approach has been adopted in [1] and the exercise indicates that a network trained by the backpropagation algorithm may not distribute the solution across all the weights. Some of the weights in the network are indeed critical and the loss of these can cause the network to fail, contradicting the requirement of fault tolerance. Simply stated, an MLPN exhibits fault tolerance if the information content of the network, captured in the connection weights, is uniformly distributed, i.e., no single node or weight is critical to the performance of the network. In this paper we discuss synthesis procedures that lead to such an appropriate choice of the network weights.

The problem of designing a fault-tolerant neural network has only recently begun to receive some attention. Neti *et al.* [11] are among the first to study this problem. They consider a standard MLPN and attempt to minimize the maximum deviation from the desired output for each input in the presence of single unit failures. The problem is thus formulated as a *constrained minimax optimization* problem. They then replace

Manuscript received February 24, 1996; revised September 30, 1997 and April 19, 1998.

D. Deodhare and M. Vidyasagar are with the Centre for Artificial Intelligence and Robotics, High Grounds, Bangalore 560 001, India.

S. Sathiya Keerthi is with the Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560 012, India.

Publisher Item Identifier S 1045-9227(98)05339-9.

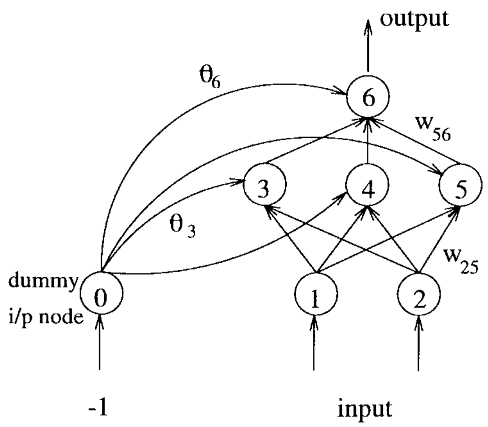

Fig. 1. A feedforward neural network.

this problem by another problem which is easier to solve; however, there is no guarantee that the solution of the modified problem is close to that of the original problem.

The main difficulty with a minimax optimization problem is that the objective function is in general *nondifferentiable*; hence gradient-based methods cannot be used to solve such problems. In the present paper, two distinct approaches are adopted to get around this difficulty. In the first approach, the constrained minimax optimization problem is replaced by a *sequence* of problems, each with a differentiable objective function; it can be shown that the corresponding sequence of solutions converges to a solution of the original minimax problem. In the second approach, the nondifferentiable objective function of the minimax problem is replaced by another differentiable objective function which is a *uniformly close approximation* to the original objective function; hence minimizing the modified objective function (which can be achieved using gradient-based methods) is *guaranteed* to be a near-optimal solution of the original minimax optimization problem.

## II. Problem Formulation

A feedforward neural network (FFNN) is conventionally represented as an acyclic directed graph denoted by $N = (V, A)$, where $V$ is the set of nodes and $A \subset V \times V$ the set of directed arcs or edges of the graph. Our notation is that $(i, j) \in A$ if and only if there is an edge *from* node $i$ *to* node $j$. A weight $w_{ij}$ is associated with every directed arc $(i, j)$ belonging to the set $A$, and a threshold value $\theta_i$ is associated with each node $i$ in a hidden layer or the output layer of the network. These threshold values can be treated in a manner similar to the weights by assuming that their negative values are connection weights on arcs from a dummy input node with an auxiliary constant-valued input of $-1$. (See Fig 1.)

Let $W$ denote the weight matrix of the network where the $ij$th entry of the matrix is $w_{ij}$. If the arc $(i, j)$ is not in the set $A$ then $w_{ij}$ is taken to be zero by convention. Further let

| | |
|---|---|
| $n$ | Number of nodes in the input layer. |
| $m$ | Number of nodes in the output layer. |
| $h$ | Number of nodes in the hidden layer(s). |
| $V_h$ | Set of hidden nodes. |
| $p$ | Number of training patterns for the network. |
| $(\boldsymbol{x}^l, \boldsymbol{d}^l)$, $l = 1, \cdots, p$ | Set of training patterns for the network, where $x^l \in \Re^n$ and $d^l \in \Re^m$. |
| $E^l$ | Error function for pattern $l, l = 1, \cdots, p$. |
| $E$ | Total error. |
| $d_j^l$ | Desired output for pattern $l$ on output node $j$. |
| $y_j^l$ | Actual output for pattern $l$ on output node $j$. |

The functioning of the network $N$ with weight matrix $W$, denoted for convenience by $N(W)$, can be described as follows.

- The value of a node in the input layer is simply the input applied to it.
- Each node $j \in V$ computes its output value after the nodes in the preceding layers have computed theirs. If we use a sigmoidal nonlinearity the rule is

$$x_j = \frac{1}{1 + \exp(-\sigma_j)}, \qquad \text{where} \quad \sigma_j = \sum_{(i,j) \in A} w_{ij} x_i.$$

This can be represented by a mapping $M(\cdot, W)$ such that $y^l = M(x^l, W)$ is the *m-dimensional* vector output of the network $N(W)$ on the input $x^l$. Let $E^l(W)$ be a suitably defined error function between the actual output of the network $y^l$ and the corresponding desired output $d^l$ on an input $x^l$. It is common to choose

$$E^l(W) = \|d^l - y^l\|_2^2$$

where the $l_2$ norm is defined as

$$\|\boldsymbol{x}\|_2^2 = \sum_{a=1}^{m} (x_a)^2.$$

However, we define it here as

$$E^l(W) = \|d^l - y^l\|_\infty$$

where the $l_\infty$ norm is defined as

$$\|\boldsymbol{x}\|_\infty = \max_{1 \le a \le m} |x_a|.$$

For our problem the $l_\infty$ norm is more meaningful since we want *each* of the values $|d^l - y^{l,i}|$ to be minimized and this requirement is more adequately captured by the $l_\infty$ norm as compared to the $l_2$ norm. The overall error is defined as

$$E(W) = \max_{l=1,\cdots,p} E^l(W).$$

Obviously, $E(W) = 0$ would imply that for each input pattern $x^l, l = 1, \cdots, p$, the network output equals the corresponding desired output vector $d^l$.

For a given node $i$ in a hidden layer of the neural network $N(W)$, let $N(W^i)$ be the network derived from $N(W)$ by removing node $i$ from $N$ and all weights corresponding to arcs starting from node $i$. (See Fig. 2.) Let $M(\cdot, W^i)$ be the new input–output mapping describing the network. If $y^{l,i} = M(x^l, W^i)$, then the error function after removing the hidden node $i$ and its associated weights is

$$E(W^i) = \max_{l=1,\cdots,p} E^l(W^i), \quad \text{where}$$
$$E^l(W^i) = \|d^l - y^{l,i}\|_\infty. \tag{1}$$

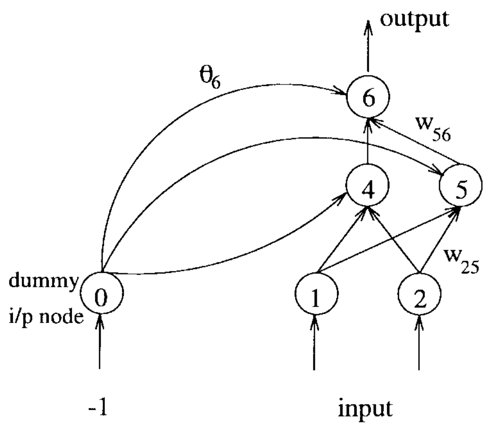

Fig. 2. $N(W^3)$: Network obtained after removing node 3 from the network in Fig. 1.

Our objective is to determine a weight matrix $W^*$ such that the network not only classifies the patterns as desired but is also *maximally* fault tolerant. Such a weight vector is defined as a solution to the constrained minimax optimization problem

$$\begin{aligned}&\min_{W}\max_{i\in V_h} E(W^i)\\&\text{subject to the constraints}\\&d^l - y^l = 0, \qquad \forall l = 1,\cdots,p.\end{aligned}\tag{2}$$

Here the quantity $E(W^i)$ represents the error in the network output when a hidden node $i$ is removed. Since we are interested in finding a weight configuration that minimizes $E(W^i)$ for all nodes $i$ in $V_h$, the problem can be naturally modeled as a constrained minimax optimization problem. This is because minimization of the *maximum* of $E(W^i)$ implies minimization of *each* of the $E(W^i)$. The hard constraints capture the requirement that when *all* the nodes in the network are functional, for each input $x^l$ to the network the output $y^l$ should equal the corresponding desired output $d^l$. Using (1) this problem can be rewritten as

$$\begin{aligned}&\min_{W}\max_{l,i,k}\max(d_k^l - y_k^{l,i}, y_k^{l,i} - d_k^l)\\&\text{subject to the constraints}\\&d^l - y^l = 0, \qquad \forall l = 1,\cdots,p\end{aligned}\tag{3}$$

where $l = 1,\cdots,p, i = 1,\cdots,h, k = 1,\cdots,m$.

Therefore to obtain a maximally fault tolerant network we need to solve a constrained minimax optimization problem. Two approaches for solving such problems are described in the next two sections.

## III. Solving the Minimax Problem—the First Approach

A general constrained minimax problem can be stated as follows.

*Problem 1:*

$$\begin{aligned}&\min_{\boldsymbol{x}}\{\boldsymbol{F}(\boldsymbol{x}) := \max_{i\in I} f_i(\boldsymbol{x})\}\\&\text{subject to the constraints}\\&g_j(\boldsymbol{x}) \le 0, \qquad j\in J,\\&h_l(\boldsymbol{x}) = 0, \qquad l\in L\end{aligned}\tag{4}$$

where $I = \{1, 2,\cdots,n\}$ is a finite set of integers.

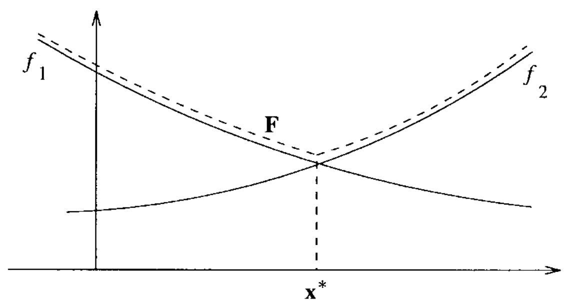

Fig. 3. In minimax optimization the objective function is not differentiable at the solution point in general.

Minimax optimization can be described as a worst-case optimization in the sense that at each iteration the algorithm has to try and reduce the function $f_i(\boldsymbol{x})$ such that $\boldsymbol{F}(\boldsymbol{x}) = f_i(\boldsymbol{x})$. Note that even if each of the functions $f_i(\boldsymbol{x})$ is differentiable, $\boldsymbol{F}(\boldsymbol{x})$ is nondifferentiable in general. The fact that $\boldsymbol{F}(\boldsymbol{x})$ is differentiable "almost everywhere" does not simplify the problem since often a solution occurs at an $\boldsymbol{x}$ where $\boldsymbol{F}(\boldsymbol{x})$ is *not* differenti able. (See Fig 3.) Therefore it is not possible to solve a minimax problem directly by using gradient-based methods, as such methods require that the objective function be differentiable. To overcome this difficulty, the constrained minimax problem is converted to a *sequence* of unconstrained minimization problems where each objective function is continuously differentiable. An efficient gradient minimization technique can then be applied to carry out the unconstrained minimization at each step of the sequence. The conversion algorithm is due to Dutta and Vidyasagar [2] and is described below.

A general constrained minimax problem is stated in (4). Now consider the objective function

$$P(\boldsymbol{x},\phi) = \sum_{i\in I(x)}[f_i(\boldsymbol{x}) - \phi]^2 + \sum_{j\in J(x)} w_j g_j^2(\boldsymbol{x}) + \sum_{l\in L} v_l h_l^2(\boldsymbol{x})$$

where $\phi$ is a prespecified constant, $w_j, j\in J$ and $v_l, l\in L$ are prespecified weights, and

$$\begin{aligned}I(\boldsymbol{x}) &= \{i\in I: f_i(\boldsymbol{x}) > \phi\}\\J(\boldsymbol{x}) &= \{j\in J: g_j(\boldsymbol{x}) > 0\}.\end{aligned}$$

By convention, empty sums are taken to equal zero. It can be shown that, for each constant $\phi, P(\boldsymbol{x},\phi)$ is continuously differentiable with respect to $\boldsymbol{x}$.

Let $\overline{\boldsymbol{x}}$ denote a solution to Problem 1. Choose an initial constant $\phi_0 \le \boldsymbol{F}(\overline{\boldsymbol{x}})$ and minimize

$$\begin{aligned}P(\boldsymbol{x},\phi_t) = &\sum_{i\in I(x)}[f_i(\boldsymbol{x}) - \phi_t]^2 + \sum_{j\in J(x)} w_j g_j^2(\boldsymbol{x})\\&+ \sum_{l\in L} v_l h_l^2(\boldsymbol{x}).\end{aligned}\tag{5}$$

Let $\overline{\boldsymbol{x}}_t$ denote a minimizer of $P(\boldsymbol{x},\phi_t)$, and update the constant $\phi$ according to the formula

$$\phi_{t+1} = \phi_t + [P(\overline{\boldsymbol{x}}_t,\phi_t)/n]^{1/2}$$

where $n$ is the total number of functions $f_i(\cdot)$. It can be shown that if $\phi_0 \le \boldsymbol{F}(\overline{\boldsymbol{x}})$, then $\phi_t \le \boldsymbol{F}(\overline{\boldsymbol{x}})$ for all $t$. Thus $\{\phi_t\}$ is a nondecreasing sequence that is bounded above, and hence has a definite limit as $t \to \infty$. Under relatively mild assumptions it can be shown [2] that $\phi_t \to \boldsymbol{F}(\overline{\boldsymbol{x}})$ as $t \to \infty$. This leads us to the following algorithm.

*Algorithm*

*Step 1)* Choose $\phi_0 \leq \boldsymbol{F}(\overline{\boldsymbol{x}})$, where $\overline{\boldsymbol{x}}$ is a solution to Problem 1. Set $t = 0$.

*Step 2)* Minimize $P(\boldsymbol{x}, \phi_t)$. Denote the solution by $\overline{\boldsymbol{x}}_t$.

*Step 3)* Set $\phi_{t+1} = \phi_t + [P(\overline{\boldsymbol{x}}_t, \phi_t)/n]^{1/2}$.

*Step 4)* If $\phi_{t+1} - \phi_t < \epsilon$, where $\epsilon$ is a prespecified small number, STOP. Otherwise, set $t \to t+1$ and go to Step 2).

To apply this algorithm to the fault tolerance problem in (3), define $e_k^{l,i} = d_k^l - y_k^{l,i}$, and

$$f(e_k^{l,i}, \phi) = \begin{cases} 0, & \text{if } e_k^{l,i} \leq \phi \\ e_k^{l,i} - \phi, & \text{if } e_k^{l,i} > \phi. \end{cases}$$

Finally

$$\begin{aligned} P(W, \phi_t) = &\sum_{i=1}^{h}\sum_{l=1}^{p}\sum_{k=1}^{m}(f^2(e_k^{l,i}, \phi_t) + f^2(-e_k^{l,i}, \phi_t)) \\ &+ \sum_{l=1}^{p}\sum_{k=1}^{m}[\mu_k^l \cdot (d_k^l - y_k^l)]^2. \end{aligned} \tag{6}$$

Here $\mu^l$ are predefined weights. The algorithm given above can now be used to minimize (6). This is referred to as **version 1** of the problem in Table III, Section VI-A. Here the second term corresponds to the third term in (5). Since the fault tolerance problem as stated has no inequality constraints there is no term in (6) corresponding to the second term in (5).

## IV. Solving the Minimax Problem—the Second Approach

An algorithm proposed by Kreisselmeier and Steinhauser [7]-[9] to solve a control-system design problem as a multiobjective optimization problem can be adapted to solve the minimax optimization problem in (3). We begin by introducing some weights into the objective function in (3), as follows:

$$\min_W\{F(W) := \max_{l,i,k} \mu_k^{l,i} \max(d_k^l - y_k^{l,i}, y_k^{l,i} - d_k^l)\}$$

where $l = 1, \cdots, p, i = 0, \cdots, h, k = 1, \cdots, m$.

Here $i = 0$ implies that the entire network is functioning, i.e., no nodes have been removed from the network. This problem differs from (3) in two respects.

- New variables $\mu_k^{l,i}$ have been introduced. Their significance is explained below.
- The equality constraints have been included in the objective function. Notationally this has been taken care of by letting $i$ range from zero to h.

If we let $\sigma$ denote the total number of functions over which the leftmost maximum is taken, then

$$\sigma = p(h+1)m.$$

Let the index $s$ range over these functions, which are written as $f_s(W)$ for convenience. Then the problem at hand becomes

$$\min_W\{F(W) := \max_{1 \leq s \leq \sigma} \mu_s f_s(W)\}. \tag{7}$$

Note that the $\mu_s$ are simply *large numbers* modeled after "penalty functions." Generally speaking, an optimization algorithm used to solve (7) deals with one index $s$ at a time—the one for which the product $\mu_s f_s(W)$ is currently the largest. Therefore by choosing appropriate $\mu_s$ sufficiently large one can force the equality constraints to be satisfied.

We now proceed to discuss the algorithm. The problem in (7) is a nonsmooth optimization problem, since the function $F(W)$ is not differentiable everywhere; but it can be approximated well by another problem with a smooth objective function. Consider the function

$$G(W) = \frac{1}{\rho}\ \ln\left\{\sum_{s=1}^{\sigma}\exp[\rho\mu_s f_s(W)]\right\} \tag{8}$$

where $\rho$ is a parameter to be specified. Note that $G(W)$ is a differentiable function of the parameter vector $W$ and can be rewritten as

$$G(W) = F(W) + \frac{1}{\rho}\ \ln\left\{\sum_{s=1}^{\sigma}\exp[\rho(\mu_s f_s(W) - F(W))]\right\}.$$

Now

$$\mu_s f_s(W) - F(W) \leq 0$$

so that

$$G(W) \leq F(W) + \frac{\ln\ \sigma}{\rho}.$$

Also, for some index $t$

$$\mu_t f_t(W) - F(W) = 0$$

and hence

$$\begin{aligned} G(W) &> F(W) + \frac{1}{\rho}\ \ln[1+\delta], \qquad \delta \geq 0 \\ &\geq F(W) \end{aligned} \tag{9}$$

where

$$\delta = \sum_{\substack{s=1 \\ s \neq t}}^{\sigma}\exp[\rho\mu_s f_s(W)].$$

So we have

$$F(W) \leq G(W) \leq F(W) + \frac{\ln\sigma}{\rho}.$$

This implies that $F(W)$ is approximated very well by $G(W)$ if $\rho \gg \ln\sigma$. Hence, the minimax problem (7) can be approximated by the smooth unconstrained problem

$$\min_W G(W).$$

## V. Improvements and Modifications

In this section we discuss a few ways of exploiting the special structure of the minimax optimization problem for neural networks, so as to facilitate the application of the algorithms described in Sections III and IV.

### A. Computing the Objective Function

A couple of simple observations help in significantly reducing the time required to compute the objective function $E(W^i)$ and hence the overall computation time.

To compute $E(W^i)$ the value of node $i \in V_h$ and all weights $w_{ij}$, where $j$ indexes over all nodes in the layer immediately above the node $i$, are taken to be zero. Note that removing node $i$ from the network does not change the values of nodes in the layer to which node $i$ belongs and those lying below it. Hence, after having computed the values of all nodes for the full network, in order to compute $E(W^i)$ we need only to recompute the values of nodes in layers above node $i$. Also to recompute the values of those nodes $j$ in the layer immediately above node $i$ a more efficient rule can be devised. Let $\sigma_j$ be the weighted sum of inputs to node $j$ when the full network is functioning. Let $x_i$ be the value of node $i$ and $a = w_{ij}x_i$. If we let $x_j(i)$ denote the value of node $j$ after removing node $i$ we have

$$x_j(i) = \frac{1}{1+\exp(-(\sigma_j - a))}.$$

Now write

$$\frac{1}{1+\exp(-(\sigma_j - a))} = \frac{1}{1+\exp(-\sigma_j)} \cdot \frac{1}{\psi}.$$

The value of $\psi$ has to be determined. Note that $1/(1+\exp(-\sigma_j)) = x_j$, the value of node $j$ when the full network is functioning. Simple algebraic manipulations lead to

$$\psi = x_j(1-\exp(a)) + \exp(a).$$

This gives

$$x_j(i) = \frac{1}{1+\exp(-(\sigma_j - a))} = \frac{x_j}{x_j(1-\exp(a))+\exp(a)}.$$

This expression obviates the necessity of recomputing the entire weighted sum of inputs to node $j$ after removing node $i$ from the network. It suffices to compute the single product $a$. The implementation of the algorithm that incorporates this improved method of computing the objective function is referred to as **version 2** in Table III, Section VI-A.

### B. Modifying the Error Function

If $x_j$ is the output value of node $j$ of the network then the derivative of the sigmoid function with respect to this output value is given by $x_j(1-x_j)$. As a result, for nodes whose output values are close to zero or one, the derivative is close to zero. Thus, in the case of a node whose output is at the wrong end of the sigmoid (i.e., where the difference between the actual output and the desired output is near one), the gradient of the error with respect to various weights in the network will be essentially zero. Consequently, if a gradient-based minimization technique is used to minimize the error, then the weight vector will essentially be "stuck" in this unacceptable location. For the *output* nodes, since the desired output is known, it is possible to introduce a new error measure so that its gradient is far from zero even when $x_j \simeq 0.0$ or $x_j \simeq 1.0$. This is done next.

With the notation given in Section II, we have

$$x_j = \frac{1}{1+\exp(-\sigma_j)}$$

where

$$\sigma_j = \sum_i w_{ij}x_i$$

denotes the weighted sum of inputs to the node $j$, and the index $i$ ranges over all the units in the layer immediately below the layer to which unit $j$ belongs.

Depending on the desired output $d_j$ for the output node $j$ two cases can be identified which are dealt with separately. First consider the case where the desired output $d_j$ equals 1.0. In this case the new error function is denoted by $r(x_j)$, and should be defined so that minimizing $r(x_j)$ effectively pushes $x_j \in (0.0, 1.0)$ towards 1.0 (as the desired output is 1.0 in this case). Define

$$r(x_j) = x_j - \ln x_j - 1.$$

The derivative of $r$ with respect to $x_j$ is

$$r'(x_j) = 1 - \frac{1}{x_j} = -\frac{(1-x_j)}{x_j}.$$

By the chain rule, the gradient of $r$ with respect to some weight $w$ equals

$$\frac{\partial r}{\partial w} = r'(x_j)\cdot[x_j(1-x_j)]\cdot\frac{\partial x_j}{\partial w} = -(1-x_j)^2\cdot\frac{\partial x_j}{\partial w}.$$

Hence $\partial r/\partial w$ will approximately equal $\partial x_j/\partial w$ if $x_j \simeq 0$. Now consider the case where the desired output $d_j$ equals 0.0. In this case we simply replace the value $x_j$ by $(1-x_j)$ in the definition of $r(x_j)$ and the rest of the discussion in case 1 holds. This is because pushing $(1-x_j)$ towards 1.0 effectively results in pushing $x_j$ towards 0.0, the desired output.

Although in the above discussion, the description of the error function has been restricted to binary valued output functions, it can be appropriately scaled to handle real-valued functions as well.

The new error function can be used only for the output units as the desired output is known only for these units. However the same problems are experienced for the hidden units. A simple solution for this is due to Fahlman [4]. In this method the derivative of the sigmoid i.e., $x_j(1-x_j)$ is altered by adding a small constant value to it. Typically this value is 0.1 so that the derivative of the sigmoid is now a curve going from 0.1 to 0.35 and back to 0.1 instead of a curve that goes from 0.0 to 0.25 to 0.0. This simple modification helps in reducing the time for convergence. The implementation of the algorithm that incorporates the above defined error function is referred to as **version 3** in Table III, Section VI-A.

## C. The Threshold and Margin Criterion

The fault tolerance problem as stated requires that, when the entire network is functioning, each output be very close to the specified target value. This is reflected in the form of equality constraints in the problem definition. (See Section II.) For networks performing a task with binary outputs this seems unnecessarily strict: sometimes the algorithm may produce useful outputs quickly, but take much longer to adjust the output values to within the specified tolerances.

The "threshold and margin" criterion [4] can be stated as follows: if the total range of output units is 0.0 to 1.0, any value below 0.4 is considered to be a zero and any value above 0.6 is considered to be a one; values between 0.4 and 0.6 are considered to be "marginal" and are not counted as correct. If we consider the value $(1 - x_j)$ instead of $x_j$ whenever $d_j = 0.0$ as suggested in the second case of Section V-B the fault tolerance problem can be modified as follows to include this criterion:

$$\min_W \{\max_{i \in V_h} E(W^i)\}$$

subject to the constraints

$$d^l - y^l \leq 0.4, \qquad \forall l \in \{i \in \{1, \cdots, p\}: d^i = 1\}$$
$$y^l - d^l \leq 0.4, \qquad \forall l \in \{i \in \{1, \cdots, p\}: d^i = 0\}. \quad (10)$$

The optimization problem of (10) is referred to as **version 4** in Table III, Section VI-A.

## VI. Computational Results and Conclusions

The above approaches were tested on three standard problems from the literature, namely, the XOR problem, the negation problem and the sonar benchmark. It is useful to discuss the XOR problem since it is the simplest problem requiring hidden units, and since many other difficult problems involve an XOR as a subproblem [13]. A network with a single hidden layer and three hidden nodes was used for the XOR problem. The network had two input nodes, and one output node as required. Consequently the corresponding optimization problem involved a total of 13 parameters—nine weights and four threshold values. In the negation problem the input to the system consists of patterns of $n + 1$ binary values and an output of $n$ values. One of the input bits is special and is known as the *negation bit.* When the negation bit equals zero the desired output consists of the last $n$ bits of the input. On the other hand, when the negation bit equals one the complement of the remaining $n$ bits is the desired output. Usually the leftmost bit of the input pattern is taken to be the negation bit. The system has no way of knowing this and must *learn* which bit is the negation bit. The negation problem with $n = 3$ was considered. The training set therefore consisted of 16 patterns. The network consisted of four input nodes, a single hidden layer with seven hidden nodes, and three output nodes. The optimization problem in this case comprised 59 parameters—49 weights and ten threshold values. Note that both the XOR problem and the negation problem are difficult problems in the sense that very similar patterns (which differ only in a single bit) require widely different outputs. To assess the performance of the algorithms proposed here on typical real-world problems the sonar benchmark was selected. A direct empirical comparison between the approach of Neti *et al.* and the first algorithm has been carried out by running both the algorithms on the sonar data and the results have been reported in Section VI-C.

TABLE I
Sample Run of the First Algorithm for the XOR Problem. Weights Were Initialized to Small Random Values, and $\mu^l = 1000.0\ \forall l$

| *Iteration* | $\phi$ | $P(W, \phi)$ |
|---|---|---|
| 1. | 0.0000 | 1.0000067302282 |
| 2. | 0.5773 | 0.17863578938668 |
| 3. | 0.8213 | $3.191 \times 10^{-2}$ |
| 4. | 0.9245 | $5.703 \times 10^{-3}$ |
| 5. | 0.9681 | $1.021 \times 10^{-3}$ |
| 6. | 0.9665 | $1.851 \times 10^{-4}$ |

| *Desired output* | *Actual output* | *Output on removing a node* | | |
|---|---|---|---|---|
| | | 1 | 2 | 3 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | *1* | 0 |

## A. The First Approach

To carry out the unconstrained minimization at each step a routine from the MINPACK library [10] was used. The subroutine minimizes the sum of the squares of multivariate scalar-valued functions by a modification of the Levenberg–Marquardt algorithm. For both the problems the weight matrix was initialized to random values. The hard (equality) constraints in the problem formulation represent the requirement that the full network (without the loss of any node) learns the input–output set of patterns as desired. This has been transformed into an unconstrained minimization problem by using the concept of penalty functions. $\mu^l, l = 1, \cdots p$ are the set of weights or penalty functions used here. [Refer to (6).] These are simply large numbers to be chosen appropriately so that the algorithm converges to the right solution. There is no obvious choice for these numbers, and the choice is usually guided by trial and error. This may appear to be a daunting task as there are $pm$ such numbers to be chosen where $p$ is the number of patterns in the pattern set and $m$ the number of nodes in the output layer. However, on the problems tested, this task turned out to be fairly simple.

- The same value was chosen for all the numbers.
- For the XOR problem all the numbers were set to the value 1000.0.
- For the negation problem all numbers were set to 1.0.

For the XOR problem, with $\mu^l = 1000.0\ \forall l$ the solution (set of weights) to which the algorithm converged had the following properties (a sample run of the program is given as Table I).

1) The network is totally tolerant to the removal of two of the three hidden nodes (removed one at a time).
2) When the third hidden node is removed, the network maps three of the four input patterns to the correct output.

TABLE II
SAMPLE RUN OF THE FIRST ALGORITHM FOR THE NEGATION PROBLEM. WEIGHTS WERE INITIALIZED TO SMALL RANDOM VALUES $\mu^l = 1.0\ \forall l$

| *Desired output* | *Actual output* | *Output on removing a node* | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 |
| 0 0 1 | 0 0 1 | 0 0 1 | 0 0 1 | 0 0 *0* | 0 0 1 | 0 0 1 | 0 0 1 | 0 0 1 |
| 0 1 0 | 0 1 0 | 0 1 0 | 0 1 0 | 0 1 0 | 0 *0* 0 | 0 1 0 | 0 1 0 | 0 1 0 |
| 0 1 1 | 0 1 1 | 0 1 1 | 0 1 1 | 0 1 *0* | 0 *0* 1 | 0 1 1 | 0 1 1 | 0 1 1 |
| 1 0 0 | 1 0 0 | 1 0 0 | 1 0 0 | 1 0 0 | 1 0 0 | *0* 0 0 | 1 0 0 | 1 0 0 |
| 1 0 1 | 1 0 1 | 1 0 1 | 1 0 1 | 1 0 1 | 1 0 1 | *0* 0 1 | 1 0 1 | 1 0 1 |
| 1 1 0 | 1 1 0 | 1 1 0 | 1 1 0 | 1 1 0 | 1 *0* 0 | 1 1 0 | 1 1 0 | 1 1 0 |
| 1 1 1 | 1 1 1 | 1 1 1 | 1 1 1 | 1 1 1 | 1 *0* 1 | 1 1 1 | 1 1 1 | 1 1 1 |
| 1 1 1 | 1 1 1 | *0* 1 1 | 1 1 1 | 1 1 1 | 1 1 1 | 1 1 1 | 1 1 *0* | 1 1 1 |
| 1 1 0 | 1 1 0 | *0* 1 0 | 1 1 0 | 1 1 0 | 1 1 0 | 1 1 0 | 1 1 0 | 1 1 0 |
| 1 0 1 | 1 0 1 | 1 0 1 | 1 0 1 | 1 0 1 | 1 0 1 | 1 0 1 | 1 0 1 | 1 0 1 |
| 1 0 0 | 1 0 0 | 1 0 0 | 1 0 0 | 1 0 0 | 1 0 0 | 1 0 0 | 1 0 0 | 1 0 0 |
| 0 1 1 | 0 1 1 | 0 1 1 | 0 1 1 | 0 1 1 | 0 1 1 | 0 1 1 | 0 1 *0* | 0 1 1 |
| 0 1 0 | 0 1 0 | 0 1 0 | 0 1 0 | 0 1 0 | 0 1 0 | 0 1 0 | 0 1 0 | 0 1 0 |
| 0 0 1 | 0 0 1 | 0 0 1 | 0 0 1 | 0 0 1 | 0 0 1 | 0 0 1 | 0 0 1 | 0 0 1 |
| 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 |

For the negation problem no penalty weights were used, i.e., $\mu^l = 1\ \forall l$. The results (see Table II) can be summarized as follows.

1) The network is totally tolerant to the removal of nodes 2 and 7.
2) There is a single bit error on two inputs when nodes 1, 3, 5, and 6 are removed.
3) There is a single bit error on four inputs when node 4 is removed.

To make the tables (Tables I and II) conveniently readable, each output value has been rounded off to the nearest zero or one value. (The errors are in italics.)

The first implementation of the algorithm (which has been called version **1**), was modified to incorporate the improvements suggested in Section V to get version **2**, version **3** and version **4**. Although all program development was done on a SPARC Station with the UNIX operating system, to compare the performance of the various versions in terms of termination time, the more convenient single-user PC-DOS environment was used. Microsoft FORTRAN 5.0 was used to compile the programs. Each version was run a number of times to solve the XOR problem with weights and thresholds initialized randomly. The starting time and terminating time were obtained from the system and the difference noted. The average was taken over 20 runs. The results are listed in Table III. Average run time comparisons between various versions justify the successive modifications made in the algorithm.

## B. Implementation of the Alternative Approach

The alternative approach to solving the minimax optimization problem discussed in Section IV was also implemented separately. Recall that the first approach of Section III to solving the minimax optimization problem involves solving a sequence of unconstrained minimization problems. The alternative approach of Section IV requires the solution of a *single* unconstrained minimization problem. Therefore in general the second algorithm is faster than the first. The subroutine used to perform the unconstrained minimization is due to Shanno and Phua [14] and minimizes an unconstrained nonlinear scalar valued function of a vector variable either by the BFGS (Broyden–Fletcher–Goldfarb–Shanno) method or by the conjugate gradient method. Obviously, the quality of the solution will depend on the minimization technique applied. Using the above mentioned routine the program performs well for the XOR problem and the output of the network is similar to that given in Table I. For the negation problem the best weight configuration to which the algorithm converged gave at least 19 bit errors in the output—a performance worse than that of the first algorithm which converged to weight configurations that gave a total of only 12 bit errors. (See Table II.) This can be explained by the fact that the routine offered by MINPACK (refer to Section VI-A) performs a minimization of the sum of the squares of nonlinear functions. While minimizing this sum it also attempts to keep the value of each of the component functions low. This leads to superior results since for the fault tolerance problem we are interested in reducing the error in each function. On the other hand, the algorithm due to Shanno and Phua simply minimizes the objective function without bothering about the component functions [i.e., $f_s(W)$—see (8)]. Consequently, when a large number of functions contribute to the objective function the results can be mediocre. However, this need not necessarily mean that the second approach is inferior. The MINPACK package is based on what are known as *trust region methods* [5] and although the package itself is tailored to solve least-squares optimization problems in theory the implementation can be extended to handle more generic optimization problems as well. The authors are not aware of any such implementation but feel that usage of such an implementation to perform the unconstrained optimization in the second algorithm may possibly lead to better results.

TABLE III
COMPARISONS BETWEEN VERSION **1**, VERSION **2**, VERSION **3**, AND VERSION **4**. THE AVERAGE WAS TAKEN OVER 20 RUNS

| *Version* | *Time taken for worst case* | *Time taken for best case* | *Average over runs* |
|---|---|---|---|
| **1** | 1 min 22.99 sec | 1 min 19.13 sec | 1 min 19.30 sec |
| **2** | 54.31 sec | 53.01 sec | 53.61 sec |
| **3** | 38.77 sec | 34.28 sec | 38.07 sec |
| **4** | 36.96 sec | 32.96 sec | 34.87 sec |

## C. Comparison with Neti et al.'s Method

In this section we present a direct empirical comparison of the first method described in Section III and that of Neti *et al.* on the sonar database. This is the data set used by Gorman and Sejnowski [6] in their study of the classification of sonar signals using a neural network. The task is to train a network to discriminate between sonar signals bounced off a metal cylinder and those bounced off a roughly cylindrical rock. The sonar benchmark comprises two files. The file "sonar.mines" contains 111 signals corresponding to sonar returns from metal cylinders. The file "sonar.rocks" contains

TABLE IV
PERFORMANCE OF TWO HIDDEN NODE NETWORKS DESIGNED USING THE FIRST METHOD

| $\mu^l$ | % correct on | | % correct on removing a node | | | |
|---|---|---|---|---|---|---|
| | | | 1 | | 2 | |
| | *trng.* | *test* | *trng.* | *test* | *trng.* | *test* |
| 0.1 | 98.1% | 84.5% | 98.1% | 86.4% | 98.1% | 86.4% |
| 1.0 | 100% | 84.5% | 100% | 83.5% | 50% | 50% |

97 signals corresponding to sonar returns from rocks. In [6], Gorman and Sejnowski discuss two series of experiments: an "aspect-angle independent" series, in which the whole data set is used without controlling for aspect angle, and an "aspect-angle dependent" series in which the training and testing sets were carefully controlled to ensure that each set contained cases from each aspect angle in appropriate proportions. Here we have used the aspect-angle dependent sets of data for training and testing the networks. Since the input signals are 60-dimensional, networks with 60 input units and two output units, one indicating a cylinder and the other a rock have been used.

The networks were initialized to small random weights using a random number generator. Several different starting points in the weight space were tried and only the best results are being reported for both the algorithms. For this the seed to the random number generator was varied. The performance of networks designed using the method of Section III on the sonar benchmark is tabulated in Table IV. For these results the seed to the random number generator was ten. The number of hidden nodes in the networks was two, and $\mu^l = 0.1\ \forall l$ for the first table entry, $\mu^l = 1.0\ \forall l$ for the second table entry.

The fault tolerance problem formulated in [11] is as follows:

$$\min_W E(W)$$
subject to the constraints
$$E(W^i) - E(W) \le \epsilon, \qquad \text{for } i \in V_h. \tag{11}$$

The aim of the problem formulation of (11) is to minimize the overall error in the network output as measured by the error $E(W)$, subject to the constraint that removing any single node from the hidden layer must not increase the error by more than $\epsilon$. In the experiments (11) is solved for a range of values of $\epsilon$ and the smallest $\epsilon$ value for which the corresponding optimal value of the objective function is close to zero is approximately determined. A successive quadratic programming algorithm from the IMSL software library has been used in [11]. The same has been done here.

Neti *et al.*'s algorithm was also run on the sonar benchmark and the results are tabulated in Tables V and VI. For these results the seed to the random number generator was 25. Since on an average the solution obtained was better for networks with three hidden nodes, in Table V the performance of three-hidden node networks (in terms of percentage classification accuracy) obtained for different values of $\epsilon$ are reported.

Note that in Table V the best solution is obtained using $\epsilon = 1.0$. The effect of changing the number of hidden nodes keeping $\epsilon$ fixed at 1.0 is reported in Table VI.

TABLE V
PERFORMANCE OF THREE HIDDEN NODES NETWORKS DESIGNED USING NETI *et al.*'S METHOD FOR DIFFERENT VALUES OF $\epsilon$

| $\epsilon$ | % correct on trng. set | % correct on removing a node | | |
|---|---|---|---|---|
| | | 1 | 2 | 3 |
| 0.0 | 50% | 50% | 50% | 50% |
| 0.1 | 88.6% | 50% | 88.6% | 88.6% |
| 0.2 | 87.6% | 50% | 85.7% | 86.7% |
| 0.3 | 81% | 50% | 83.9% | 81.9% |
| 0.33 | 71.4% | 50% | 74.3% | 66.7% |
| 0.5 | 78.1% | 50% | 74.3% | 64.8% |
| 0.75 | 86.7% | 50% | 77.1% | 72.4% |
| 0.88 | 89.5% | 50% | 88.6% | 82.9% |
| 0.92 | 91.4% | 50% | 87.6% | 85.7% |
| 0.95 | 94.3% | 50% | 89.5% | 87.6% |
| 1.0 | 95.2% | 50% | 95.2% | 90.5% |

Since the solutions obtained for network configurations with three hidden nodes and four hidden nodes were the best, only the performance of these networks on the test set have been tabulated in Table VII. For easy reference the results reported in [6] for the angle-dependent experiments are being reproduced here partially in Table VIII. The performance of our algorithm compares favorably both with the results obtained by Neti *et al.*'s method and those obtained in [6].

The second entry in Table IV indicates that even with one node functioning (i.e., node 2 in the two hidden nodes network) the network is able to achieve 100% classification accuracy on the training set. This suggested that perhaps the sonar database consists of a linearly separable set of data points. To test this a routine from the neural networks toolbox of MATLAB was used to train a perceptron on the entire set of data points (208 in all). The perceptron training algorithm converged in 1 172 480 epochs confirming that the sonar data is in fact linearly separable.[1] To assess the perfo rmance of our algorithm on "realistic" *not* linearly separable data the sonar data was perturbed a little so that it was no longer linearly separable. For this the data points in the rocks file and the data points in the mines file were moved toward each other along the line joining the respective centroids of these points. The performance of the algorithm on this new data is given in Table IX.

## VII. DISCUSSION

Currently used learning algorithms result in nonuniform weights and thresholds, of which a few are critical and many others are insignificant. Therefore there is a need for new approaches that yield specific weights and thresholds that incorporate fault tolerance. This work is a step in this direction. At this point it is important to make some general comments regarding the solutions obtained by the methods discussed.

[1]Note that the experiments of Gorman and Sejnowski with zero hidden nodes did not demonstrate the linear separability of the data. This is because minimizing the least-squares error function is not the same as minimizing the number of misclassifications.

TABLE VI
PERFORMANCE OF 2, 3, 4, 6 HIDDEN NODE NETWORKS DESIGNED USING NETI *et al.*'S METHOD WITH $\epsilon = 1$

| *Hidden nodes* | $\epsilon$ | *% correct on trng. set* | *% correct on removing a node* | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | 1.0 | 50% | 50% | 50% | – | – | – | – |
| 3 | 1.0 | 95.2% | 50% | 95.2% | 90.5% | – | – | – |
| 4 | 1.0 | 94.3% | 50% | 94.3% | 94.3% | 94.3% | | – |
| 6 | 1.0 | < 10% | 50% | 50% | < 50% | 50% | 50% | 50% |

TABLE VII
PERFORMANCE OF 3, 4 HIDDEN NODE NETWORKS DESIGNED USING NETI *et al.*'S METHOD WITH $\epsilon = 1$ ON THE TEST SET

| *Hidden nodes* | $\epsilon$ | *% correct on test set* | *Output on removing a node* | | | |
|---|---|---|---|---|---|---|
| | | | 1 | 2 | 3 | 4 |
| 3 | 1.0 | 79.6% | 50% | 79.6% | 78.1% | – |
| 4 | 1.0 | 79% | 50% | 79% | 80% | 80% |

TABLE VIII
RESULTS FOR THE ANGLE-DEPENDENT EXPERIMENTS

| *Hidden nodes* | *% correct on training set* | *% correct on test set* |
|---|---|---|
| 0 | 79.3 | 73.1 |
| 2 | 96.2 | 85.7 |
| 3 | 98.1 | 87.6 |
| 6 | 99.4 | 89.3 |
| 12 | 99.8 | 90.4 |
| 24 | 100.0 | 89.2 |

TABLE IX
PERFORMANCE OF TWO THREE-HIDDEN-NODE NETWORKS DESIGNED USING THE FIRST METHOD WITH $\mu^l = 1.0\ \forall l$ ON THE *perturbed* SONAR DATA

| *Hidden nodes* | $\mu^l$ | *% correct on* | | *% correct on removing a node* | | | |
|---|---|---|---|---|---|---|---|
| | | | | 1 | | 2 | |
| | | *trng.* | *test* | *trng.* | *test* | *trng.* | *test* |
| 2 | 1.0 | 99% | 77.6% | 99% | 75.7% | 50% | 50% |
| 3 | 1.0 | 99% | 75.7% | 99% | 75.7% | 99% | 75.7% |

### *A. How Good Are the Solutions Obtained?*

The solutions obtained can be described as good. The programs were run for several times with a different set of starting weights and the best results have been reported. Although it may be possible to get a better solution for the negation problem, our intuitive feeling is that it may not be possible to improve on the solution for the XOR problem. The fact that the XOR problem cannot be solved by a single hidden node MLP may prohibit this. These observations are based merely on an empirical study, and more analysis needs to be carried out in order to make any definite claims. The performance of the first algorithm on the sonar benchmark is obviously very good.

### *B. How Simple Is the Network Design Used?*

When we speak of the simplicity of the network we refer to the number of hidden nodes, as the number of input nodes and the number of output nodes are fixed by the set of input and output patterns. For the XOR problem the choice of three hidden nodes is the least possible, since on a single node fault, we need at least two functioning nodes in order for the network to produce the correct output. An important point to note here is that even when the number of hidden nodes was increased from three up to eight one at a time, the single bit error in the output persisted. A possible explanation for this phenomenon as given by one of the reviewers is as follows: In the usual solution to the XOR problem obtained by training a two-hidden node network using backpropagation one node acts as an OR-gate while the other acts as a NAND-gate. The NAND-gate inhibits the "1"-output when both inputs are "1". Since the NAND-gate is only functional for the two-"1's" input, it is less critical as compared to the OR-gate. For this very reason, it is possible that the effect of the constrained optimization is to replicate the "more important" OR-gate. Then one could remove any of these replicates and the remaining nodes would ensure correct network operation. But removing the (single) NAND-gate will cause the network to fail for the two-"1's" input.

For the negation problem a network with two hidden nodes was used to start with and this number was increased one at a time. An acceptable solution was finally obtained when the network architecture had seven hidden nodes. It is not obvious whether the following fact is relevant but a negation problem can be reduced to a set of three XOR's between the negation bit and each input.

### *C. How Easy Is It to Incorporate Fault Tolerance?*

When one compares between the effort required to train the network by an existing conventional method (like backpropagation for example) and that required to incorporate fault tolerance as described here, it is found that the latter requires substantially more effort. In other words, both in terms of time and space, incorporating fault tolerance is significantly more expensive. The problem mainly arises due to the choice of using conventional optimization algorithms to perform the minimization. These need to be modified or tailored to suit the size of a neural-network problem.

## VIII. CONCLUSIONS

Some important lessons have been learned through this research. These can be listed as follows.

1) The fault tolerance requirement for a neural network needs to be incorporated in the problem formulation it-

self and training algorithms need to be designed keeping this requirement in mind.

2) Most sophisticated optimization algorithms make use of some sort of approximate inverse Hessian. This matrix is of very large size, and is difficult to store for large networks. Therefore efficiently applying quasi-Newton methods to large neural networks is an important problem. We need methods that are efficient both in space and time. Besides, to conform to the neural-network approach they also need to be local and parallel. Some work in this direction has been reported by Pearlmutter [12].

The methods proposed here lead to networks that exhibit a *partial* degree of fault tolerance. However it may be possible to extend these methods to ensure *guaranteed* fault tolerance. Guaranteed fault tolerance implies that the network continues to function after the removal of *any* of the hidden nodes of the network. A simple strategy to achieve this can be based on replication. Once network weights using minimax optimization are obtained, critical weights of the network can be identified and replicated. This approach may prove to be less expensive than a purely replication-based approach. Other constructive methods need to be explored.

We conclude with the comment that the extra expense of embedding fault tolerance arises only during the training phase of the network. Once a robust set of weights has been determined for a given problem, the on-line functioning of the network in terms of time and space is not significantly affected. In most applications of artificial neural networks, training is an off-line process and hence these methods can find practical use in spite of the added cost.

## Acknowledgment

The authors are grateful to the reviewers for their comments. These have vastly helped in improving the paper.

## References

[1] G. Bolt, J. Austin, and G. Morgan, "Fault-tolerant multilayer perceptron networks," Dept. Comput. Sci., Univ. York, Hestington, York, U.K., Tech. Rep. YCS 180, July 1992.

[2] S. R. K. Dutta and M. Vidyasagar, "New algorithms for constrained minimax optimization," *Math. Programming*, vol. 13, no. 2, pp. 140–155, Oct. 1977.

[3] M. D. Emmerson and R. I. Damper, "Determining and improving the fault tolerance of multilayer perceptrons in a pattern-recognition application," *IEEE Trans. Neural Networks*, vol. 4, pp. 788–793, Sept. 1993.

[4] S. E. Fahlman, "An empirical study of learning speed in backpropagation networks," June 1988.

[5] R. Fletcher, *Practical Methods of Optimization*. New York: Wiley-Interscience, 1987.

[6] R. P. Gorman and T. J. Sejnowski, "Analysis of hidden units in a layered network trained to classify sonar targets," *Neural Networks*, vol. 1, pp. 75–89, 1988.

[7] J. M. Maciejowski, *Multivariable Feedback Design*. Reading, MA: Addison-Wesley, 1989.

[8] G. Kreisselmeier and R. Steinhauser, "Systematic control design by optimizing a vector performance index," in *Proc. IFAC Symp. Comput.-Aided Design Contr. Syst.*, Zurich, Switzerland, 1979, pp. 113–117.

[9] ——, "Application of vector performance optimization to a robust control loop design for a fighter aircraft," *Int. J. Contr.*, vol. 37, pp. 251–284, 1983.

[10] J. J. Moré, D. C. Sorensen, B. S. Garbow, and K. E. Hillstorm, *MINPACK Project*, Argonne National Laboratory, Mar. 1980.

[11] C. Neti, M. H. Schneider, and E. D. Young, "Maximally fault-tolerant neural networks," *IEEE Trans. Neural Networks*, vol. 3, pp. 4–23, Jan. 1992.

[12] B. A. Pearlmutter, "Fast multiplication by the hessian," *Neural Comput.*, June 1993.

[13] D. E. Rumelhart and J. L. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1. Cambridge, MA: MIT Press, 1986.

[14] D. F. Shanno and K. H. Phua, "Minimization of unconstrained multivariate functions," *ACM Trans. Math. Software*, vol. 6, Dec. 1980, pp. 618–622.

**Dipti Deodhare** received the M.Sc. degree in computer science from Pune University, India. She received the M.S. (Engg.) degree in computer science and automation from Indian Institute of Science, Bangalore.

She has been with the Centre for Artificial Intelligence and Robotics, Bangalore since March 1991.

**M. Vidyasagar** (S'69–M'69–SM'78–F'83) was born in Guntur, Andhra Pradesh, India, on September 29, 1947. He received the B.S., M.S., and Ph.D. degrees, all in electrical engineering, from the University of Wisconsin, Madison, in 1965, 1967, and 1969, respectively.

He has taught at Marquette University, Milwaukee , from 1969 to 1970, Concordia University, Montreal, Canada, from 1970 to 1980, and the University of Waterloo, Ontario, Canada, from 1980 to 1989. Since June 1989, he has been the Director of the Centre for Artificial Intelligence and Robotics (under the Defence Research and Development Organization), Bangalore, India. In addition to the above, he has held visiting positions at several universities including the Massachusetts Institute of Technology, the University of California (Berkeley, Los Angeles), C.N.R.S., Toulouse, France, the Indian Institute of Science, the University of Minnesota, Minneapolis, and Tokyo Institute of Technology, Japan. He is the author or coauthor of seven books and more than 120 papers in archival journals.

Dr. Vidyasagar has received several honors in recognition of his research activities, including the Distinguished Service Cit ation from his Alma Mater (The University of Wisconsin). In addition, he is a Fellow of the Indian Academy of Sciences, the Indian National Science Academy, the Indian National Academy of Engineering, and the Third World Academy of Sciences.

**S. Sathiya Keerthi** received the Ph.D degree in control engineering from The University of Michigan, Ann Arbor, in 1987.

Since April 1987 he has been with the faculty of the Department of Computer Science and Automation, Indian Institute of Science, where he is currently an Associate Professor. His main research interests include optimization and geometric algorithms, neural networks, and robot path planning.