M. A. Sartori and P. J. Antsaklis, "Neural Network Training via Quadratic Optimization," Technical Report #90-05-01, Dept. of Electrical and Computer Engineering, University of Notre Dame, May 1990. Revised April 1991.

NEURAL NETWORK TRAINING VIA QUADRATIC OPTIMIZATION

Technical Report #90-05-01
Department of Electrical Engineering
University of Notre Dame
May 1990
Revised April 1991

Michael A. Sartori and Panos J. Antsaklis
Department of Electrical Engineering
University of Notre Dame
Notre Dame, Indiana 46556

M. A. Sartori and P. J. Antsaklis, "Neural Network Training via Quadratic Optimization," Technical Report #90-05-01, Dept. of Electrical and Computer Engineering, University of Notre Dame, May 1990. Revised April 1991.

NEURAL NETWORK TRAINING VIA QUADRATIC OPTIMIZATION

Michael A. Sartori and Panos J. Antsaklis
Department of Electrical Engineering
University of Notre Dame
Notre Dame, Indiana 46556

ABSTRACT

A new technique using quadratic optimization is proposed to find the weights of a single neuron, or a single-layer neural network, and extended to the multi-layer neural network. It is proposed here to find the weights for a neuron by minimizing a cost function that is quadratic with respect to the neuron's weights and to use these weights as an answer for minimizing a cost function that is quadratic with respect to the neuron's outputs. A careful error analysis of this procedure is provided. Previous methods, such as the least mean squares algorithm which is a gradient descent method and a precursor of the back-propagation algorithm, iteratively find weights for the neuron which minimize the cost function directly involving the nonlinearity of the neuron. By back-propagating the output error through the neural network's layers, the proposed method is extended to the multi-layer neural network. The described Quadratic Optimization Algorithm for the multi-layer neural network tends to work best for classification problems and tends to achieve successful results in a single iteration.

A new training method based on quadratic optimization is presented in this paper to find the weights of a single neuron, or a single-layer neural network, and is extended to a multi-layer neural network. Instead of minimizing a cost function that directly involves the nonlinearity of the neuron, a function which is quadratic with respect to the neuron's weights is minimized. The solution from this minimization problem is used as a solution for the original problem, and the relationship between the two problems is established through a careful error analysis and an examination of the relationship between the minima. Due to the class of nonlinear functions often chosen for the neuron (e.g., the hyperbolic tangent function or the signum function), the error for using the solution from the quadratic minimization as a solution for the original problem is small, and even zero if the error from solving the quadratic minimization is zero, which is a case studied

here. Furthermore, with the quadratic optimization procedure used to find the weights of the single neuron, it always converges and is faster than using a gradient descent algorithm on the original problem. By back-propagating the output error through a multi-layer neural network's hidden layers, the proposed method is extended to the multi-layer case. The so-called Quadratic Optimization Algorithm for the multi-layer neural network tends to work best for classification problems and tends to achieve successful results in a single iteration.

In Section 1, the neuron considered in this paper and the problem of finding its weights, termed here the Neuron Training Problem (N), are defined. In Section 2, the Neuron Quadratic Optimization Problem (NQ) is defined, and its solution is used as one for Problem (N). Including the special case of zero error, an error analysis is conducted for this usage, and the relationship between the minima of (NQ) and those of (N) is examined. In Section 3, the single-layer neural network and the Single-Layer Neural Network Training Problem (L) are defined. The quadratic optimization procedure is then described for the single-layer neural network in terms of the Problem (L) and termed the Single-Layer Neural Network Quadratic Optimization Problem (LQ). In Section 4, the multi-layer neural network and the Multi-Layer Neural Network Training Problem (M) are defined, and the back-propagation algorithm, one of the most common methods used to train the multi-layer neural network, is described. In Section 5, it is proposed to solve the Problem (LQ) for each layer of the multi-layer neural network by back-propagating the output layer's error to each hidden layer. The resulting procedure is termed the Quadratic Optimization Algorithm. Finally, in Section 6, examples are given that illustrate the training procedure of this paper.

1 THE NEURON

The neuron considered here is described by

$$y = f(\sum_{i=1}^{m} u_i\, w_i) = f(\mathbf{u}'\mathbf{w}), \tag{1}$$

where $f:\mathbb{R} \rightarrow \mathbb{R}$ is the nonlinearity of the neuron, $\mathbf{u} := [u_1, ..., u_m]' \in \mathbb{R}^{mx1}$ is the input vector, $\mathbf{w} := [w_1, ..., w_m]' \in \mathbb{R}^{mx1}$ is the weight vector, and $u_m = 1$ is the bias input for the neuron. The type of nonlinearity of the neuron is restricted to those functions commonly used in neuron models (e.g., the hyperbolic tangent function or the signum function).

Assume that a training set $\{\mathbf{u}(j), d(j)\}$ for $1 \leq j \leq p$ consists of p pairs of input vectors and desired output scalars, where $\mathbf{u}(j) \in \mathbb{R}^{mx1}$, $u_m(j) = 1$, and $d(j) \in \mathbb{R}$ for $1 \leq j \leq p$. The Neuron Training Problem (N) is defined as follows:

where

$$\left.\begin{aligned} &\min_{\mathbf{w}} \hat{F}(\mathbf{w}) \\ &\hat{F}(\mathbf{w}) = (\mathbf{d} - \phi(\mathbf{U}'\mathbf{w}))'(\mathbf{d} - \phi(\mathbf{U}'\mathbf{w})) \end{aligned}\right\} \quad (N)$$

and where $\mathbf{d} := [d(1), ..., d(p)]' \in \mathbb{R}^{p\times 1}$ is the desired output vector, $\mathbf{U} := [\mathbf{u}(1), ..., \mathbf{u}(p)] \in \mathbb{R}^{m\times p}$ is the matrix of input vectors, and $\phi(\mathbf{z}) := [f(z_1), ..., f(z_p)]' \in \mathbb{R}^{p\times 1}$ with $\mathbf{z} = [z_1, ..., z_p]' \in \mathbb{R}^{p\times 1}$. The notation $\phi(\mathbf{z})$ represents a map which takes a p-dimensional vector $\mathbf{z}$ and returns another p-dimensional vector with element $f(z_i)$, where f is the neuron's nonlinearity. In equation (N), $\hat{F}(\mathbf{w})$ is actually the sum of the squares of the error between the desired scalars and the output of the neuron:

$$\hat{F}(\mathbf{w}) = \sum_{j=1}^{p} (d(j) - f(\mathbf{u}(j)'\mathbf{w}))^2. \quad (2)$$

If the popular gradient descent algorithm is used to solve (N), an iterative update equation of the form

$$w_i(t+1) = w_i(t) - \alpha \frac{\delta \hat{F}(\mathbf{w})}{\delta w_i} \quad (3)$$

is applied where t denotes the iteration number of the algorithm. If the nonlinearity of the neuron is continuously differentiable, (3) is equivalent to

$$w_i(t+1) = w_i(t) + 2\alpha \sum_{j=1}^{p} (d(j) - y(j))\, u_i(j) \frac{\delta f(\mathbf{u}(j)'\mathbf{w})}{\delta \mathbf{u}(j)'\mathbf{w}} \quad (4)$$

which is the update equation of the back-propagation algorithm for the output layer of the multi-layer neural network. In general, the gradient descent algorithm does not guarantee convergence to a global minimum due to the potential local minima entrapment [Gill81, Bazaraa79]. If a gradient descent algorithm is applied to a quadratic function, it converges at a linear rate if a line search is used to find the step length at each iteration. If a gradient descent algorithm is applied to a general function and not a quadratic one, the algorithm is potentially very slow. In addition, at regions of very low gradient, a gradient descent algorithm takes small orthogonal steps which result in a "zigzagging" effect of the updates and slow convergence [Gill81, Bazaraa79].

If the signum function is used as the neuron's nonlinearity, the gradient descent algorithm can not be applied to (N) directly since the signum function is not differentiable. Applying the results of [Widrow60] to solve (N), a slightly different cost function is used since the signum function is assumed to be the neuron's nonlinearity:

$$\hat{F}(\mathbf{w}) = \sum_{j=1}^{p} (d(j) - \mathbf{u}(j)'\mathbf{w})^2. \quad (5)$$

With this, the following gradient descent rule results:

$$w_i(t+1) = w_i(t) + 2\alpha \sum_{j=1}^{p} (d(j) - \mathbf{u}(j)'\mathbf{w})u_i(j). \quad (6)$$

In [Rosenblatt62], a signum is once again assumed to be the nonlinearity of the neuron, and the proposed update equation for the weights is

$$w_i(t+1) = w_i(t) + 2\alpha \sum_{j=1}^{p} (d(j) - f(\mathbf{u}(j)'\mathbf{w}))u_i(j). \tag{7}$$

In [Shynk90], various cost functions are described for which (7) is a gradient descent rule. These described methods are gradient descent procedures and hence suffer the problems of all gradient descent procedures.

2 QUADRATIC PROBLEM FORMULATION AND ANALYSIS

Since the Neuron Training Problem (N) is actually an unconstrained minimization problem, a variety of optimization techniques exist which may be employed to solve it. Unfortunately, due to the type of nonlinearities which are usually chosen for this problem (for example, the hyperbolic tangent function), the surface of the function $\hat{F}(\mathbf{w})$ is, in general, very complicated, and finding a $\mathbf{w}$ which minimizes the surface may be a very difficult task. It is proposed here that instead of finding a $\mathbf{w}$ which minimizes $\hat{F}(\mathbf{w})$, solve the following Problem (NQ) and use the solution of (NQ) as an answer for (N). The Neuron Quadratic Optimization Problem (NQ) is defined as follows:

where
$$\left.\begin{aligned} &\min_{\mathbf{w}} F(\mathbf{w}) \\ &F(\mathbf{w}) = (\mathbf{v} - \mathbf{U}'\mathbf{w})'(\mathbf{v} - \mathbf{U}'\mathbf{w}) \end{aligned}\right\} \tag{NQ}$$

and where $\mathbf{v}$ is such that $\phi(\mathbf{v}) = \mathbf{d}$. The function $F(\mathbf{w})$ can be re-written as:

$$F(\mathbf{w}) = \mathbf{w}'\mathbf{A}\mathbf{w} - \mathbf{h}'\mathbf{w} + c, \tag{8}$$

where $\mathbf{A} := \mathbf{U}\mathbf{U}' \in \mathbb{R}^{m \times m}$, $\mathbf{h}' := 2\mathbf{v}'\mathbf{U}' \in \mathbb{R}^{1 \times m}$, and $c := \mathbf{v}'\mathbf{v} \in \mathbb{R}^{1 \times 1}$. So, finding a $\mathbf{w}$ that minimizes $F(\mathbf{w})$ in (NQ) is equivalent to finding a $\mathbf{w}$ that minimizes (8).

For the general quadratic function

$$G(\mathbf{w}) = \mathbf{w}'\mathbf{A}\mathbf{w} - \mathbf{h}'\mathbf{w} + c, \tag{9}$$

let $\mathbf{A} \in \mathbb{R}^{m \times m}$ be symmetric and positive semi-definite, $\mathbf{h}' \in \mathbb{R}^{1 \times m}$, and $c \in \mathbb{R}^{1 \times 1}$. If $\mathbf{A}$ is positive definite and rank[$\mathbf{A}$] = m, $G(\mathbf{w})$ is a quadratic function which possesses only one minimum which is the global minimum. The vector $\mathbf{w}$ is the minimum of $G(\mathbf{w})$ if

$$\nabla G(\mathbf{w}) = \mathbf{0}. \tag{10}$$

Setting the gradient of (9) equal to zero, the vector $\mathbf{w}$ is the minimum of $G(\mathbf{w})$ if $\mathbf{w}$ solves

$$2\mathbf{A}\mathbf{w} = \mathbf{h}. \tag{11}$$

Denoting the solution to (11) as

$$\mathbf{w}^* := \frac{1}{2}\mathbf{A}^{-1}\mathbf{h}, \tag{12}$$

the minimum of $G(\mathbf{w})$ is given by

$$G(\mathbf{w}^*) = c - \frac{1}{4}\mathbf{h}'\mathbf{A}^{-1}\mathbf{h}. \tag{13}$$

Now, applying this to the Problem (NQ),

$$\nabla F(\mathbf{w}) = \mathbf{0} \tag{14}$$

implies that

$$\mathbf{U}\mathbf{U}'\mathbf{w} = \mathbf{U}\mathbf{v}. \tag{15}$$

Note that if $\mathbf{w}$ satisfies

$$\mathbf{U}'\mathbf{w} = \mathbf{v}, \tag{16}$$

it always satisfies (15). However, (15) does not necessarily imply (16) unless rank[$\mathbf{U}'$] = p ($\leq$ m), which is Case (i) below. If rank[$\mathbf{U}'$] = m, then $\mathbf{A} = \mathbf{U}'\mathbf{U}$ is positive definite, and $F(\mathbf{w})$ possesses only one minimum which is the global minimum. The relationship between this minimum and the minimum of $\hat{F}(\mathbf{w})$ is discussed in Section 2.2.

Next, the following two cases are addressed: Case (i) when there are at least as many weights as patterns, that is m $\geq$ p, and Case (ii) when there are more patterns than weights, that is p > m. For Case (i) with m $\geq$ p and rank[$\mathbf{U}'$] = p, the weights $\mathbf{w}^*$ of the neuron can be found by solving (16), which in this case is equivalent to solving (15) or (14). Note that such solutions as $\mathbf{w}^*$ always exist. Furthermore, since $\mathbf{U}'\mathbf{w}^* = \mathbf{v}$, $F(\mathbf{w}^*) = 0$; in view of (NQ), the minimum in this case is zero. To solve (16), many methods can be employed, and one possible solution is given by

$$\mathbf{w}^* = \mathbf{U}^+\mathbf{v}, \tag{17}$$

where $\mathbf{U}^+$ is the pseudo-inverse of $\mathbf{U}'$. Using the singular value decomposition of $\mathbf{U}'$, $\mathbf{w}^*$ can be computed. For the case of m $\geq$ p, the singular value decomposition of $\mathbf{U}'$ is

$$\mathbf{C}^H\mathbf{U}'\mathbf{D} = \begin{bmatrix}\Sigma & 0\end{bmatrix} \tag{18}$$

where Σ is a diagonal matrix of singular values, $\mathbf{C}$ and $\mathbf{D}$ are unitary matrices, and $\mathbf{C}^H$ denotes the Hermitian, or conjugate transpose, of $\mathbf{C}$. After some manipulation, the solution is

$$\mathbf{w}^* = \mathbf{D}\begin{bmatrix}\Sigma^{-1} \\ \mathbf{0}\end{bmatrix}\mathbf{C}^H\mathbf{v}. \tag{19}$$

The weight vector $\mathbf{w}^*$ of (19) minimizes

$$\|\mathbf{v} - \mathbf{U}'\mathbf{w}\|_2^2. \tag{20}$$

Furthermore, for any other $\tilde{\mathbf{w}}$ which also minimizes (20), $\|\mathbf{w}^*\|_2 < \|\tilde{\mathbf{w}}\|_2$.

For Case (ii) with p > m and rank[$\mathbf{U}'$] = m and in view of (15), there exists a $\mathbf{w}^*$ such that $\nabla F(\mathbf{w}^*) = \mathbf{0}$, but $F(\mathbf{w}^*)$ may not be zero since in this case $\mathbf{U}'\mathbf{w}^* \neq \mathbf{v}$ in general. With the solution of (15) denoted as

$$\mathbf{w}^* = \frac{1}{2}[\mathbf{U}\mathbf{U}']^{-1}\mathbf{U}\mathbf{v} = \mathbf{U}^+\mathbf{v}, \tag{21}$$

(NQ) is solved, and the minimum of $F(\mathbf{w})$ is given in this case by

$$F(\mathbf{w}^*) = \mathbf{v}'[\mathbf{I}_p - \mathbf{U}'\mathbf{U}^+]\mathbf{v}, \tag{22}$$

which is not necessarily zero. There exist many ways to solve (15) and compute $\mathbf{w}^*$ of (21) since it is a system of linear algebraic equations, which is a problem that has been extensively studied. For instance, the computation of $\mathbf{w}^*$ can be performed using the conjugate gradient algorithm [Bertsekas89], which converges in a number of steps less than or equal to the number of weights in the neuron.

Solving (NQ) is, in general, easier than solving (N). It is proposed here to solve (NQ) instead of (N) and to use the solution from (NQ) as an answer for (N). Intuitively speaking, if the $\mathbf{w}^*$ found from solving (NQ) also solves (N) with a small error, then this validates minimizing $F(\mathbf{w})$ instead of minimizing $\hat{F}(\mathbf{w})$.

2.1 Error Analysis

The error for using the solution $\mathbf{w}$ from solving (NQ) as an answer for (N) is quantified in this section.

Definition 2.1:

For any $\mathbf{w}$, the error $\underline{\varepsilon} \in \mathbb{R}^{p\times 1}$ in Problem (NQ) is defined as

$$\underline{\varepsilon} := \mathbf{v} - \mathbf{U}'\mathbf{w}. \tag{23}$$

Definition 2.2:

For any $\mathbf{w}$, the error $\hat{\underline{\varepsilon}} \in \mathbb{R}^{p\times 1}$ in Problem (N) is defined as

$$\hat{\underline{\varepsilon}} := \mathbf{d} - \phi(\mathbf{U}'\mathbf{w}). \tag{24}$$

The errors $\underline{\varepsilon}$ and $\hat{\underline{\varepsilon}}$ are both p-dimensional vectors where

$$\varepsilon(j) = v(j) - \mathbf{u}(j)'\mathbf{w} \tag{25}$$

for $1 \le j \le p$ and

$$\hat{\varepsilon}(j) = d(j) - f(\mathbf{u}(j)'\mathbf{w}) \tag{26}$$

for $1 \le j \le p$. Furthermore,

$$\underline{\varepsilon}'\underline{\varepsilon} = (\mathbf{v} - \mathbf{U}'\mathbf{w})'(\mathbf{v} - \mathbf{U}'\mathbf{w}) = F(\mathbf{w}) \tag{27}$$

and

$$\hat{\underline{\varepsilon}}'\hat{\underline{\varepsilon}} = (\mathbf{d} - \phi(\mathbf{U}'\mathbf{w}))'(\mathbf{d} - \phi(\mathbf{U}'\mathbf{w})) = \hat{F}(\mathbf{w}). \tag{28}$$

The next theorems, corollaries, and lemma address the relationship between the errors for Problems (N) and (NQ).

Theorem 2.1:

The errors in Problems (N) and (NQ), $\hat{\underline{\varepsilon}}$ and $\underline{\varepsilon}$ respectively, are related by

$$\hat{\underline{\varepsilon}} = \phi(\mathbf{U}'\mathbf{w} + \underline{\varepsilon}) - \phi(\mathbf{U}'\mathbf{w}), \tag{29}$$

where

$$\hat{\varepsilon}(j) = f(\mathbf{u}(j)'\mathbf{w} + \varepsilon(j)) - f(\mathbf{u}(j)'\mathbf{w}) \tag{30}$$

for $1 \leq j \leq p$.

Proof:

From (23),

$$\mathbf{v} = \mathbf{U}'\mathbf{w} + \underline{\varepsilon}.$$

Applying the nonlinearity f to both sides,

$$\phi(\mathbf{v}) = \mathbf{d} = \phi(\mathbf{U}'\mathbf{w} + \underline{\varepsilon}).$$

Substituting into (24),

$$\hat{\underline{\varepsilon}} = \phi(\mathbf{U}'\mathbf{w} + \underline{\varepsilon}) - \phi(\mathbf{U}'\mathbf{w}). \quad \blacklozenge$$

The following corollaries and theorems address the zero error case.

Corollary 2.2:

For any $\mathbf{w}$ if $\underline{\varepsilon} = \mathbf{0}$, then $\hat{\underline{\varepsilon}} = \mathbf{0}$ for the same $\mathbf{w}$.

Proof:

For a particular $\mathbf{w}$, if $\underline{\varepsilon} = \mathbf{0}$ is substituted into (29), then

$$\hat{\underline{\varepsilon}} = \phi(\mathbf{U}'\mathbf{w} + \mathbf{0}) - \phi(\mathbf{U}'\mathbf{w}) = \mathbf{0}.$$

Thus, $\hat{\underline{\varepsilon}} = \mathbf{0}$ for the same $\mathbf{w}$. $\blacklozenge$

Corollary 2.3:

Let the nonlinear function of the neuron be one-to-one. For any $\mathbf{w}$, $\hat{\underline{\varepsilon}} = \mathbf{0}$ if and only if $\underline{\varepsilon} = \mathbf{0}$ for the same $\mathbf{w}$.

Proof:

($\Rightarrow$): If $\hat{\varepsilon}(j) = 0$ for $1 \leq j \leq p$, then

$$\hat{\varepsilon}(j) = 0 = f(\mathbf{u}(j)'\mathbf{w} + \varepsilon(j)) - f(\mathbf{u}(j)'\mathbf{w}),$$
$$f(\mathbf{u}(j)'\mathbf{w} + \varepsilon(j)) = f(\mathbf{u}(j)'\mathbf{w}),$$
$$\mathbf{u}(j)'\mathbf{w} + \varepsilon(j) = \mathbf{u}(j)'\mathbf{w},$$

and $\varepsilon(j) = 0$ for $1 \leq j \leq p$.

($\Leftarrow$): Use the Proof of Corollary 2.2. $\blacklozenge$

Theorem 2.4:

Let the nonlinear function of the neuron be one-to-one. There exists a solution $\mathbf{w}$ to (N) such that $\hat{\underline{\varepsilon}} = \mathbf{0}$ if and only if rank[U':v] = rank[U']. Furthermore, this solution $\mathbf{w}$ is unique if and only if rank[U'] = m (or equivalently, the dimension of the right null space of U' is zero).

Proof:

Using Corollary 2.3 and Definition 2.1, $\hat{\underline{\varepsilon}} = \mathbf{0}$ if and only if $\underline{\varepsilon} = \mathbf{0}$ or if and only if there exists a $\mathbf{w}$ such that

$$\mathbf{U}'\mathbf{w} = \mathbf{v} \tag{16}$$

is satisfied. From the theory of linear algebraic equations, there exists a solution $\mathbf{w}$ to (16) if and only if rank[$\mathbf{U}'$:$\mathbf{v}$] = rank[$\mathbf{U}'$]. Furthermore, a solution to (16) is unique if and only if the dimension of the right null space of $\mathbf{U}'$ is zero, which is true only when rank[$\mathbf{U}'$] = m. ♦

The following Lemma presents an important relationship between the two errors, $\hat{\varepsilon}(j)$ and $\varepsilon(j)$ for $1 \le j \le p$.

Lemma 2.5:

Let the nonlinearity of the neuron be continuous and differentiable. For $1 \le j \le p$, the error $\hat{\varepsilon}(j)$ can be approximated by $\varepsilon(j)f'(\mathbf{u}(j)'\mathbf{w})$.

Proof:

If the nonlinear function of the neuron is continuous and differentiable, the derivative of $f(\mathbf{u}(j)'\mathbf{w})$ is given by

$$f'(\mathbf{u}(j)'\mathbf{w}) = \lim_{\varepsilon(j)\to 0} \frac{f(\mathbf{u}(j)'\mathbf{w} + \varepsilon(j)) - f(\mathbf{u}(j)'\mathbf{w})}{\varepsilon(j)}.$$

Using Theorem 2.1, the error $\hat{\varepsilon}(j)$ can be approximated by $\varepsilon(j)f'(\mathbf{u}(j)'\mathbf{w})$ for $1 \le j \le p$. ♦

For the types of continuous nonlinearities often chosen for the neuron (e.g., the hyperbolic tangent function and the sigmoid function), the slopes are small for arguments large in magnitude. Thus, a large magnitude for $\varepsilon(j)$ can be tolerated if $\mathbf{u}(j)'\mathbf{w}$ is located in these relatively flat regions and thereby producing a small error for (N).

2.2 The Relationship Between the Solutions of (NQ) and (N)

The relationship between the minimum of $F(\mathbf{w})$ and the minima of $\hat{F}(\mathbf{w})$ is examined in this section. The gradient of $F(\mathbf{w})$ is

$$\nabla F(\mathbf{w}) = -2\mathbf{U}\underline{\varepsilon}. \tag{31}$$

If $\nabla F(\mathbf{w}) = 0$, the weight vector $\mathbf{w}$ is the minimum of $F(\mathbf{w})$, and

$$F(\mathbf{w}) = \mathbf{v}'\underline{\varepsilon}. \tag{32}$$

This zero gradient can occur if $\mathbf{U}\underline{\varepsilon} = \mathbf{0}$. This can of course happen when $\underline{\varepsilon} = \mathbf{0}$, which was the case studied in Section 2.1. In addition, this is satisfied if the error $\underline{\varepsilon}$ is in the right null space of $\mathbf{U}$, or if

$$\sum_{j=1}^{p} u_k(j)\varepsilon(j) = 0 \tag{33}$$

for $1 \le k \le m$, where m is the number of weights and p is the number of patterns.

The gradient of $\hat{F}(\mathbf{w})$ is

$$\nabla\hat{F}(\mathbf{w}) = -2\mathbf{U}\ \mathrm{diag}(\nabla(\phi(\mathbf{U}'\mathbf{w})))\ \hat{\underline{\varepsilon}} \tag{34}$$

where

$$\nabla(f(u_k(j)w_k)) = \frac{\delta f(u_k(j)w_k)}{\delta(u_k(j)w_k)} \tag{35}$$

for $1 \le j \le p$, $1 \le k \le m$. For the hyperbolic tangent function,

$$\frac{\delta f(u_k(j)w_k)}{\delta(u_k(j)w_k)} = (1 - f(u_k(j)w_k)^2), \tag{36}$$

and for the sigmoid function ($f(z) = 1/(1 - e^{-z})$),

$$\frac{\delta f(u_k(j)w_k)}{\delta(u_k(j)w_k)} = (f(u_k(j)w_k) - f(u_k(j)w_k)^2). \tag{37}$$

If $\nabla\hat{F}(\mathbf{w}) = 0$, the weight vector w is the minimum of $\hat{F}(\mathbf{w})$. This zero gradient can occur if

$$\sum_{j=1}^{p} u_k(j)\ \frac{\delta f(u_k(j)w_k)}{\delta(u_k(j)w_k)}\ \hat{\varepsilon}(j) = 0. \tag{38}$$

Clearly, if $\hat{\underline{\varepsilon}} = \mathbf{0}$, then $\hat{F}(\mathbf{w}) = 0$.

By comparing the conditions for $\nabla F(\mathbf{w}) = 0$ and $\nabla\hat{F}(\mathbf{w}) = 0$, the relationship between the minimum of $F(\mathbf{w})$ and the minima of $\hat{F}(\mathbf{w}) = 0$ can be studied. If $\underline{\varepsilon} = \mathbf{0}$, then $\hat{\underline{\varepsilon}} = \mathbf{0}$, as described in Corollary 2.2. This implies $F(\mathbf{w}) = 0$, $\hat{F}(\mathbf{w}) = 0$, and the weight vector w is the minimum of $F(\mathbf{w})$ and of $\hat{F}(\mathbf{w})$. If $\mathbf{U}\underline{\varepsilon} = \mathbf{0}$ but $\underline{\varepsilon} \neq \mathbf{0}$, then $\hat{\underline{\varepsilon}}$ is not necessarily zero, and the weight vector w is the minimum of $F(\mathbf{w})$ but not necessarily a minimum of $\hat{F}(\mathbf{w})$. The determination of whether w is a minimum of $\hat{F}(\mathbf{w})$ depends on the values $\frac{\delta f(u_k(j)w_k)}{\delta(u_k(j)w_k)}$. Comparing (35) and (40), if $\frac{\delta f(u_k(j)w_k)}{\delta(u_k(j)w_k)}$ $\cong$ 1, then the weight vector w is the minimum of $F(\mathbf{w})$ and very close to a minimum of $\hat{F}(\mathbf{w})$. Assuming that the neuron's nonlinearity is either the hyperbolic tangent function or the sigmoid function, this can occur if $u_k(j)w_k$ is large in magnitude, and hence located in either of the "flat" regions of the neuron's nonlinearity.

3 THE SINGLE-LAYER NEURAL NETWORK

The single-layer neural network considered here is comprised of n parallel neurons each described by

$$y_i = f(\mathbf{u}'\mathbf{w}_i) \tag{39}$$

for $1 \le i \le n$. For the i^{th} neuron, the function $f:\mathbb{R} \rightarrow \mathbb{R}$ is the neuron's nonlinearity and is restricted to those commonly used in neuron models, $\mathbf{u} := [u_1, ..., u_m]' \in \mathbb{R}^{mx1}$ is the input vector, $\mathbf{w_i} := [w_{1i}, ..., w_{mi}]' \in \mathbb{R}^{mx1}$ is the weight vector, and $u_m = 1$ is the bias input for the neuron.

Assume that a training set consisting of p pairs of input vectors and desired output vectors $\{\mathbf{u}(j), \mathbf{d}(j)\}$ for $1 \le j \le p$ is given, where $\mathbf{u}(j) \in \mathbb{R}^{mx1}$, $u_m(j) = 1$, and $\mathbf{d}(j) = [d_1(j), ..., d_n(j)]' \in \mathbb{R}^{nx1}$ for $1 \le j \le p$. The output of the single-layer neural network is described by

$$\mathbf{Y} = \Phi(\mathbf{U}'\mathbf{W}) \tag{40}$$

where $\mathbf{Y} := [\mathbf{y_1}, ..., \mathbf{y_n}]' \in \mathbb{R}^{pxn}$ is the matrix of the neural network's outputs, $\mathbf{y_i} := [y_i(1), ..., y_i(p)]' \in \mathbb{R}^{px1}$ for $1 \le i \le n$ are the vectors of a particular neuron's output, $\mathbf{U} := [\mathbf{u}(1), ..., \mathbf{u}(p)]' \in \mathbb{R}^{mxp}$ is the matrix of input vectors, $\mathbf{W} := [\mathbf{w_1}, ..., \mathbf{w_n}] \in \mathbb{R}^{mxn}$ is the matrix of weight vectors, and $\Phi(\mathbf{Z}) := [\phi(\mathbf{z_1}), ..., \phi(\mathbf{z_n})] \in \mathbb{R}^{pxn}$ with $\mathbf{Z} := [\mathbf{z_1}, ..., \mathbf{z_n}] \in \mathbb{R}^{pxn}$. The notation $\Phi(\mathbf{Z})$ represents a map which takes a matrix $\mathbf{Z}$ with elements z_{ji} and returns another matrix of the same size with elements $f(z_{ji})$, where f is the neuron's nonlinearity.

The Single Layer Neural Network Training Problem (L) is defined as follows:

where

$$\left.\begin{aligned} &\min_{\mathbf{W}} \hat{F}(\mathbf{W}) \\ &\hat{F}(\mathbf{W}) = \mathrm{tr}((\mathbf{D} - \Phi(\mathbf{U}'\mathbf{W}))'(\mathbf{D} - \Phi(\mathbf{U}'\mathbf{W}))) \end{aligned}\right\} \tag{L}$$

and where "tr" is the trace of a square matrix, $\mathbf{D} := [\mathbf{d_1}, ..., \mathbf{d_n}] \in \mathbb{R}^{pxn}$ is the matrix of desired outputs, and $\mathbf{d_i} := [d_i(1), ..., d_i(p)]' \in \mathbb{R}^{px1}$ for $1 \le i \le n$ are the desired output vectors. With n = 1, (L) reduces to (N). In equation (L), $\hat{F}(\mathbf{W})$ is actually a sum of the squares of the error between the individual desired output elements and the outputs of the neurons:

$$\hat{F}(\mathbf{W}) = \sum_{k=1}^{n} \sum_{j=1}^{p} (d_k(j) - f(\mathbf{u}(j)'\mathbf{w_k}))^2 \tag{41}$$

Since Problem (L), like Problem (N), is actually an unconstrained minimization problem, a variety of optimization techniques exist which may be employed to solve it. Due to the type of nonlinear functions which are usually chosen for the neurons, the surface of the function $\hat{F}(\mathbf{W})$ is, in general, very complicated, and finding a $\mathbf{W}$ which minimizes the surface may be a very difficult task. The iterative training methods discussed in the previous section for the neuron can be extended to find the weights for a single-layer neural network, and the problems encountered for these methods unfortunately carry over to the training of the single-layer neural network.

M. A. Sartori and P. J. Antsaklis, "Neural Network Training via Quadratic Optimization," Technical Report #90-05-01, Dept. of Electrical and Computer Engineering, University of Notre Dame, May 1990. Revised April 1991.

3.1 The Single-Layer Neural Network Quadratic Optimization Problem

It is proposed here that instead of finding a $\mathbf{W}$ which minimizes $\hat{F}(\mathbf{W})$, solve the following Problem (LQ) and use the solution of (LQ) as an answer for (L). The Single-Layer Neural Network Quadratic Optimization Problem (LQ) is defined as follows:

where
$$\left.\begin{aligned} &\min_{\mathbf{W}} F(\mathbf{W}) \\ &F(\mathbf{W}) = \mathrm{tr}((\mathbf{V} - \mathbf{U}'\mathbf{W})'(\mathbf{V} - \mathbf{U}'\mathbf{W})) \end{aligned}\right\} \quad \text{(LQ)}$$

and where $\Phi(\mathbf{V}) = \mathbf{D}$, $\mathbf{V} = [\mathbf{v}_1, ..., \mathbf{v}_n] \in \mathbb{R}^{p \times n}$, $\mathbf{v}_i := [v_i(1), ..., v_i(p)]' \in \mathbb{R}^{p \times 1}$ for $1 \le i \le n$, and $f(v_i(j)) = d_i(j)$ for $1 \le i \le n$, $1 \le j \le p$. With $n = 1$, (LQ) reduces to (NQ). The function $F(\mathbf{W})$ can be re-written as:

$$F(\mathbf{W}) = \sum_{i=1}^{n} \mathbf{w}_i'\mathbf{A}\mathbf{w}_i - \mathbf{h}_i'\mathbf{w}_i + c_i \qquad (42)$$

where $\mathbf{A} := \mathbf{U}\mathbf{U}' \in \mathbb{R}^{m \times n}$, $\mathbf{h}_i' := 2\mathbf{v}_i'\mathbf{U}' \in \mathbb{R}^{1 \times m}$, and $c_i := \mathbf{v}_i'\mathbf{v}_i \in \mathbb{R}^{1 \times 1}$ for $1 \le i \le n$. Thus, with $\mathbf{A}$ symmetric and positive definite, $F(\mathbf{W})$ is the sum of quadratics. The solving of (LQ) can be accomplished in many ways including either minimizing the n quadratics of (42) or finding a $\mathbf{W}$ that solves

$$\mathbf{U}\mathbf{U}'\mathbf{W} = \mathbf{U}\mathbf{V}. \qquad (43)$$

Either way, the previous results for the single neuron are still applicable, but the appropriate changes due to the increased sizes of $\mathbf{d}$ and $\mathbf{w}$ to $\mathbf{D}$ and $\mathbf{W}$, respectively, need to be performed.

4 THE MULTI-LAYER NEURAL NETWORK

The multi-layer neural network considered here consists of many layers of parallel neurons connected in a feedforward manner. Defining the symbol #k as the number of neurons in the k^{th} layer, the output of the k^{th} layer is described by

$$\mathbf{Y}^k = \Phi(\mathbf{U}^k\mathbf{W}^k) \qquad (44)$$

where $\mathbf{Y}^k := [\mathbf{y}_1^k, ..., \mathbf{y}_{\#k}^k] \in \mathbb{R}^{p \times \#k}$ is the matrix of outputs, $\mathbf{y}_i^k := [y_i^k(1), ..., y_i^k(p)]' \in \mathbb{R}^{p \times 1}$ is the vector of outputs for the i^{th} neuron, $\mathbf{U}^k := [\mathbf{u}^k(1), ..., \mathbf{u}^k(p)] \in \mathbb{R}^{(\#(k-1)+1) \times p}$ is the matrix of input vectors, $\mathbf{u}^k(j) := [y_1^{k-1}(j), ..., y_{\#(k-1)}^{k-1}(j), 1]' \in \mathbb{R}^{(\#(k-1)+1) \times 1}$ is the vector of inputs for the j^{th} input pattern and is equal to the outputs from the previous layer plus the bias of one for the last term, $\mathbf{W}^k := [\mathbf{w}\backslash O(_1{}^k), ..., \mathbf{w}\backslash O(_{\#k}{}^k)] \in \mathbb{R}^{(\#(k-1)+1) \times \#k}$ is the matrix of weight vectors, $\mathbf{w}_i^k := [w_{1,i}^k, ..., w_{\#(k-1)+1,i}^k]' \in \mathbb{R}^{(\#(k-1)+1) \times 1}$ is the vector of weights, and $\Phi(\mathbf{Z}) \in \mathbb{R}^{p \times n}$ is the same as defined previously for the single-layer neural network. Using $\mathbf{U}^1 = \mathbf{U}$, the output of the first hidden layer is described by

$$\mathbf{Y}^1 = \Phi(\mathbf{U}^1\mathbf{W}^1). \qquad (45)$$

With $\mathbf{U}^{2'} = [\mathbf{Y}^1\ \mathbf{1}] \in \mathbb{R}^{p\times(\#1+1)}$ where $\mathbf{1} \in \mathbb{R}^{p\times 1}$, the output of the second hidden layer is described by

$$\mathbf{Y}^2 = \Phi(\mathbf{U}^{2'}\mathbf{W}^2). \tag{46}$$

Continuing this inductive process, each successive layer is defined appropriately until the desired number of layers is reached. The last layer is called the output layer and is described by

$$\mathbf{Y}^o = \Phi(\mathbf{U}^{o'}\mathbf{W}^o) \tag{47}$$

where the superscript "o" denotes "output".

The Multi-Layer Neural Network Training Problem (M) is defined as follows:

where
$$\left.\begin{array}{l} \min\limits_{\mathbf{W}^1 \ldots \mathbf{W}^o} \hat{F}(\mathbf{W}^1, \ldots, \mathbf{W}^o) \\ \\ \hat{F}(\mathbf{W}^1, \ldots, \mathbf{W}^o) = \mathrm{tr}((\mathbf{D} - \mathbf{Y}^o)'(\mathbf{D} - \mathbf{Y}^o)) \end{array}\right\} \tag{M}$$

and where "tr" is the trace of a square matrix, $(\mathbf{W}^1, \ldots, \mathbf{W}^o)$ are the weight matrices of all the layers of the multi-layer neural network, $\mathbf{D} = [\mathbf{d}_1, \ldots, \mathbf{d}_n]' \in \mathbb{R}^{p\times n}$ is the desired output matrix, and $\mathbf{Y}^o$ is the output of the output layer of the multi-layer neural network. In relation to the previous training problem (L), the input matrix $\mathbf{U}$ is not directly in (M) since the input is "buried" beneath the hidden layers. If there are no hidden layers, then (M) reduces to (L). In equation (M), $\hat{F}(\mathbf{W}^1, \ldots, \mathbf{W}^o)$ is actually the sum of the squares of the error between the individual desired output elements and the outputs of the neurons in the output layer:

$$\hat{F}(\mathbf{W}^1, \ldots, \mathbf{W}^o) = \sum_{k=1}^{n}\sum_{j=1}^{p} (d_k(j) - y_k^o(j))^2 \tag{48}$$

One method to solve (M) is the back-propagation algorithm [Rumelhart86], which is a constant-step-size, gradient-descent algorithm that minimizes the least-squares cost function $\hat{F}(\mathbf{W}^1, \ldots, \mathbf{W}^o)$. The sigmoid function is often considered to be the nonlinear function of the neurons (i.e., $f(z) = 1/(1 + e^{-z})$) and has the property that $f'(z) = f(z)(1 - f(z))$. Here, the hyperbolic tangent function, $f(z) = \tanh(z)$, is used as the neuron's nonlinearity, and $f'(z) = 1 - f(z)^2$. The weights of the neural network are adjusted after every epoch (i.e., one pass of the training set) by the constant-step-size gradient-descent rule:

$$w_{hi}^k(t+1) = w_{hi}^k(t) - \alpha \frac{\delta \hat{F}}{\delta w_{hi}^k}. \tag{49}$$

After some manipulation of the partial derivative term, the back-propagation algorithm's rule for changing the weights of the multi-layer neural network is given by

$$w_{hi}^k(t+1) = w_{hi}^k(t) + \alpha \sum_{j=1}^{p} \delta_i^k(j)\, y_h^{k-1}(j) \tag{50}$$

where $\delta_i^k(j)$ is known as the delta term. If the k^{th} layer is the output layer, then

$$\delta_i^o(j) = [d_i(j) - y_i^o(j)]\,[1 - y_i^o(j)^2]. \tag{51}$$

For the hidden layers,

$$\delta_i^k(j) = \sum_{r=1}^{\#(k+1)} \delta_r^{k+1}(j)\, w_{ir}^{k+1}\, [1 - y_i^k(j)^2]. \tag{52}$$

5 THE QUADRATIC OPTIMIZATION ALGORITHM

In this section, the Multi-Layer Neural Network Training Problem (M) is solved by applying the solution of Problem (LQ) to each layer. The technique proposed here is based on the back-propagation algorithm, in which the propagation of the output error of the multi-layer neural network is used to form a desired output for each layer and hence to form a quadratic weight cost function at each layer. The solution of Problem (LQ) for each layer is then used as a solution for Problem (M). Several implementation considerations are discussed as well as the advantages and disadvantages of using this approach.

Instead of solving the Problem (M) using the back-propagation algorithm, it is proposed here to solve the Problem (LQ) for each layer of the multi-layer neural network and use this solution as one for (M). In solving (LQ) for the single-layer neural network, the desired output is known, and thus a matrix $\mathbf{V}$ can be found such that $\Phi(\mathbf{V}) = \mathbf{D}$. In solving (LQ) for the k^{th} layer of the multi-layer neural network, the matrix $\mathbf{V^k}$ needs to be found such that

$$\Phi(\mathbf{V^k}) = \mathbf{D^k} \tag{53}$$

where $\mathbf{V^k} := [v_1^k, \ldots, v_{\#k}^k] \in \mathbb{R}^{p \times \#k}$, $v_i^k := [v_i^k(1), \ldots, v_i^k(p)]' \in \mathbb{R}^{p \times 1}$, and $f(v_i^k(j)) = d_i^k(j)$ for $1 \le i \le \#k$, $1 \le j \le p$.

Assuming that all weights in the hidden layers have initial values, there is no problem in directly applying the methodology described for the single-layer neural network to the output layer. A matrix $\mathbf{V^0}$ can be chosen such that $\Phi(\mathbf{V^0}) = \mathbf{D}$, and Problem (LQ) can be applied to find the weights of the output layer. Unfortunately, there do not exist desired outputs $\mathbf{D^k}$ for the hidden layers, but by using the back-propagation of the output error, an approximation of these values can be obtained; the algorithm proposed here back-propagates the error between the desired output and the actual output of the neural network to all of the hidden layers to form an approximated desired output for each layer.

First, the errors at the output of each layer are defined. The error between the desired output and actual output for the i^{th} neuron of the k^{th} layer is given by

$$\hat{\varepsilon}_i^k(j) = d_i^k(j) - y_i^k(j), \tag{54}$$

where $1 \le i \le \#k$, $1 \le j \le p$, and the error for the i^{th} neuron of the output layer is the quantity

$$\hat{\varepsilon}_i^o(j) = d_i(j) - y_i^o(j), \tag{55}$$

where $1 \le i \le n$, $1 \le j \le p$. Next, in comparing the delta terms of (51) and (52) of the back-propagation algorithm, the error for the i^{th} neuron of the k^{th} layer (not equal to the output layer) can be viewed as

$$\hat{\varepsilon}_i^k(j) = \sum_{h=1}^{\#(k+1)} \delta_h^{k+1}(j)\, w_{ih}^{k+1}. \tag{56}$$

Combining (54) and (56), the desired output for the i^{th} neuron of the k^{th} layer can be viewed as

$$d_i^k(j) = y_i^k(j) + \sum_{h=1}^{\#(k+1)} \delta_h^{k+1}(j)\, w_{ih}^{k+1}. \tag{57}$$

Using (57), the matrix $\mathbf{V^k}$ can be chosen such that

$$f(v_i^k(j)) = y_i^k(j) + \sum_{h=1}^{\#(k+1)} \delta_h^{k+1}(j)\, w_{ih}^{k+1}, \tag{58}$$

where $1 \le i \le \#k$, $1 \le j \le p$. With $\mathbf{V^k}$, Problem (LQ) can be solved to find the weights of the k^{th} layer. Since the hyperbolic tangent function is one-to-one and is assumed to be the nonlinear function of each neuron, $v_i^k(j)$ can be formed by applying the inverse of the function to both sides of (58). Thus, by back-propagating the error through the multi-layer neural network, a quadratic problem is formulated for each layer, and the results for the Problem (LQ) in relation to the Problem (L) are applicable here for each layer. This method does not guarantee convergence, but does tend to give good results with a fast computation time.

In implementing this quadratic optimization procedure for a multi-layer neural network, several observations are useful. First, in practice, limiting the neural network to two layers provides adequate results. Second, since a quadratic function is minimized for each layer, the hidden layer should be adjusted first, and then the output layer can be updated using the newly found values for the hidden layer's weights. Third, when a one-to-one function is the hidden layer's nonlinearity and when the values $v_i^1(j)$ are found by inverting the one-to-one function, care must be taken to insure that $d_i^1(j)$ lies in the range of the function. For instance, if the hyperbolic tangent function is the nonlinearity for the hidden layer, its range is (-1, 1) and hence $d_i^1(j) \in$ (-1, 1). To insure this, the output layer's weights need to be first initialized to small values around zero, for instance $w_{ih}^2 \in [-\frac{1}{\#1}, \frac{1}{\#1}]$. Next, when computing the desired output for the hidden layer, if the right-hand side of (58) is not in the range of the hidden layer's nonlinearity, the weights of the output layer can be scaled to insure this: if $\bar{w}^2 = \max\{w_{ih}^2$ for $1 \le i \le \#1 + 1$ and $1 \le h \le \#2\} > 1/\#1$, then (58) is modified to

$$f(v_i^1(j)) = y_i^1(j) + \sum_{h=1}^{\#2} \delta_h^2(j)\, \frac{w_{ih}^2}{\#1\,\bar{w}^2} \tag{59}$$

for $1 \le i \le \#1 + 1$ and $1 \le j \le p$. If the right-hand side of (59) is still not in the range of the hidden layer's nonlinearity, the output layer's weights can continue to be scaled by 1/#1 until this occurs.

Fourth, in practice, the weights for the output layer tend to be large in magnitude, which is attributed to the fitting of the mapping between $\mathbf{U^2}$ and $\mathbf{V^2}$ with the linear equation

$$\mathbf{U^2 U^{2'} W^2 = U^2 V^2}. \tag{60}$$

To aid in avoiding the computational inaccuracies which may occur due to the large magnitudes of $\mathbf{W^2}$, it is suggested to choose $v_i^2(j) < 0.5$ for $1 \le i \le \#2$ and $1 \le j \le p$. This can be accomplished by scaling the desired outputs appropriately. This also aids in insuring that $d_i^1(j)$ is properly valued. With these observations, the Quadratic Optimization Algorithm used in practice to train a multi-layer neural network is as follows:

1) Given $\mathbf{D}$, find $\mathbf{V^2}$ such that $v_i^2(j) < 0.5$ for $1 \le i \le \#2$ and $1 \le j \le p$.
2) Initialize $\mathbf{W^1}$ and $\mathbf{W^2}$ with magnitudes less than 1/#1.
3) Test $\hat{F}(\mathbf{W^1}, \mathbf{W^2})$. If small enough, then stop.
4) Find $\mathbf{V^1}$ using either (58) or (59).
5) Find $\mathbf{W^1}$ by solving $\mathbf{U^1 U^{1'} W^1 = U^1 V^1}$.
6) Find $\mathbf{U^2}$ for the new weights $\mathbf{W^1}$.
7) Find $\mathbf{W^2}$ by solving $\mathbf{U^2 U^{2'} W^2 = U^2 V^2}$.
8) Goto 3).

In applying the Quadratic Optimization Algorithm to various problems, several advantages and disadvantages are evident. First, finding the weights of a multi-layer neural network via the quadratic optimization approach described here tends to work well for classification problems in that the desired outputs are achieved and the generalization behavior of the neural network is accurate. The algorithm also tends to converge to a solution in a single step achieving a small value for $\hat{F}(\mathbf{W^1}, \mathbf{W^2})$ and then to slowly vary around this value with more iterations. Thus, it is recommended to use the Quadratic Optimization Algorithm for a single iteration on classification problems.

Both of these properties are attributed to the finding of the weights via steps 5) and 7) of the algorithm. To achieve this type of performance, the choice of the number of hidden layer neurons is important. Since the overall mapping between the input patterns and the desired output patterns is accomplished via the solving of the linear system of equations in step 7), the number of hidden layer neurons needs to be large enough such that this linear approximation in the output layer succeeds. Clearly, the choice of #1 is problem dependent. Thus, the choice of the number of hidden layer neurons is a design consideration and is dependent on the particular desired mapping of the training set. Furthermore, the initial values for the weights of the neural network are more important as the number of hidden layer neurons is reduced towards the level where the Quadratic Optimization Algorithm is unable to achieve the desired training set mapping. These properties of the algorithm are illustrated in Example 6.2.

Disadvantages of using the Quadratic Optimization Algorithm to train a multi-layer neural network are outlined next. First, the algorithm may not work well for non-classification problems in that the desired outputs may be approximately achieved but the generalization behavior of the neural network may be inaccurate. This behavior is attributed to the finding of the output layer's weights by the solving of the linear system of equations in step 7). Thus, it is suggested to restrict the use of the Quadratic Optimization Algorithm to classification problems. Secondly, in applying the algorithm, all of the training patterns need to be known and no values for the weights from previous iterations are saved. Thus, this quadratic optimization training procedure may not work for on-line learning. Thirdly, the algorithm also requires the solving of two linear systems of equations when there are two layers of weights: one in step 5) with m equations and m unknowns, and the other in step 7) with #1 + 1 equations and #1 + 1 unknowns. If these numbers are large, the solving of the linear systems may become burdensome, although there do exist many ways for solving such systems. Finally, the values for the output layer weights may be large, which is a potential disadvantage if implementation of the neural network is desired. This behavior is also attributed to the final calculation step of the algorithm, which attempts to form the desired mapping with the solving of a linear system of equations to the find the weights of the neural network's output layer.

Examples 6.2 to 6.7 in the following section illustrate these observations and some of the advantages and disadvantages of using the Quadratic Optimization Algorithm to train a multi-layer neural network.

6 EXAMPLES

Example 6.1:

As shown in Figure 1, a 5x5 retina is shown with four different letters. The input patterns generated for these letters consist of either a 1 (black) or a -1 (white) and are formed by copying the elements of the retina matrix row-wise into a vector such that $\mathbf{U} \in \mathbb{R}^{26x4}$ is the input matrix. The desired scalars associated with the input patterns for T and H are assigned a 0.9, and those associated with the input patterns for E and N are assigned a -0.9 such that $\mathbf{d} \in \mathbb{R}^{4x1}$. It is desired to learn this mapping using a single neuron trained with the quadratic optimization approach of Section 3. With the hyperbolic tangent as the neuron's nonlinearity, the vector $\mathbf{v} \in \mathbb{R}^{4x1}$ has elements v(j) = atanh(d(j)) for $1 \leq j \leq p$:

$$\mathbf{v} = [1.4722 \quad 1.4722 \quad -1.4722 \quad -1.4722]'.$$

Applying Theorem 2.4, rank[U'] = 4. So, there exists a solution $\mathbf{w} \in \mathbb{R}^{26x1}$ to (N) such that $\hat{\underline{\varepsilon}} = \mathbf{0}$. Using the singular value decomposition of U', the w is computed via (19):

M. A. Sartori and P. J. Antsaklis, "Neural Network Training via Quadratic Optimization," Technical Report #90-05-01, Dept. of Electrical and Computer Engineering, University of Notre Dame, May 1990. Revised April 1991.

$$\mathbf{w} = \begin{bmatrix} 0.2236 \\ -0.1304 \\ -0.1304 \\ -0.1304 \\ 0.2236 \\ -0.3261 \\ -0.8992 \\ 0.5498 \\ 0.0000 \\ 0.3541 \\ -0.3261 \\ 0.5730 \\ 0.2236 \\ 0.5730 \\ 0.3541 \\ -0.3261 \\ 0.0000 \\ 0.5498 \\ -0.8992 \\ 0.3541 \\ -0.3261 \\ -0.6802 \\ -0.1304 \\ -0.6802 \\ -0.3261 \\ 0.2236 \end{bmatrix}$$

and

$$\mathbf{y} = [0.9 \;\; 0.9 \;\; -0.9 \;\; -0.9]' = \mathbf{d}.$$

Actually, since $p \leq m$, any set of desired outputs can be realized by a single neuron. In fact, for the given 5x5 retina, all 26 letters of the alphabet can be represented and any division of the letters can be accomplished via a single neuron. Using the quadratic optimization method presented here, this division can be immediately determined by solving a linear system of equations.

Figure 1 Retina patterns for the letters T, H, E, and N.

Example 6.2:

In this example, a comparison of the Quadratic Optimization Algorithm and the back-propagation algorithm for the training of a multi-layer neural network is presented for an extended XOR training set, where

$$U' = \begin{matrix} -1 & -1 & 1 \\ -1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & 1 \\ -2 & -2 & 1 \\ -2 & 2 & 1 \\ 2 & -2 & 1 \\ 2 & 2 & 1 \end{matrix} \in \mathbb{R}^{8x3}$$

and

$$\mathbf{d} = [0.1 \quad -0.1 \quad -0.1 \quad 0.1 \quad 0.1 \quad -0.1 \quad -0.1 \quad 0.1]' \in \mathbb{R}^{8x1}.$$

Both the Quadratic Optimization Algorithm and the back-propagation algorithm are implemented in MATLAB on a Macintosh SE using programs that are not optimized. Steps 5) and 7) of the algorithm are solved using the psuedo-inverse function call for MATLAB. Using a two-layer neural network with the hyperbolic tangent function as the nonlinearity for both the hidden layer neurons and the output layer neuron, the number of hidden layer neurons is changed.

The training results for the two algorithms are compared in terms of the success of training the neural network for the desired classification, the values for the cost function $\hat{F}$, and the number of floating point operations, which is a function call in MATLAB. The multi-layer neural network is initialized to weights in the interval $[-\frac{1}{\#1}, \frac{1}{\#1}]$, and first trained using the Quadratic Optimization Algorithm. Next, the same initial neural network is trained using the back-propagation algorithm until the same value for $\hat{F}$ is achieved. (Note that at each iteration step of the back-propagation algorithm the gradient of (50) is not computed and is instead approximated using a single training pattern.) These results are shown in Table 1. The value $\hat{F}(t)$ denotes the value of $\hat{F}(\mathbf{W}^1, w^2)$ after t iterations of the training procedure. The value "flops" indicates the number of floating point operations as counted by MATLAB for the t iterations. For the cases of 4 and 5 hidden layer neurons, the back-propagation trained neural network did not classify the input set correctly, and the training was continued until a lower $\hat{F}$ was achieved. The resulting neural networks classified the input patterns correctly, and the results for the extended training using the back-propagation algorithm are shown in Table 2. The values for t and flops are for the total training time. To illustrate the convergence of the back-propagation algorithm, the intermediate values of $\hat{F}$ for the neural network with 5 hidden layer neurons are plotted in Figure 2. As was described in the previous section, the multi-layer neural network trained with the Quadratic Optimization Algorithm requires enough hidden layer neurons such that the solution to the linear system of equations in

M. A. Sartori and P. J. Antsaklis, "Neural Network Training via Quadratic Optimization," Technical Report #90-05-01, Dept. of Electrical and Computer Engineering, University of Notre Dame, May 1990. Revised April 1991.

Table 1 Comparing the Quadratic Optimization Algorithm and the back-propagation algorithm.

	Quadratic Optimization				Back-Propagation		
#1	$\hat{F}(0)$	t	$\hat{F}(t)$	flops	t	$\hat{F}(t)$	flops
6	0.1063	1	0.0212	132024	1244	0.0208	1442790
5	0.1189	1	0.0244	10256	1169	0.0240	1158168
4	0.1330	1	0.0246	820	427	0.0240	351273

Table 2 Continuing training with the back-propagation algorithm.

#1	t	$\hat{F}(t)$	flops
5	1729	0.0093	1713385
4	1006	0.0096	476024

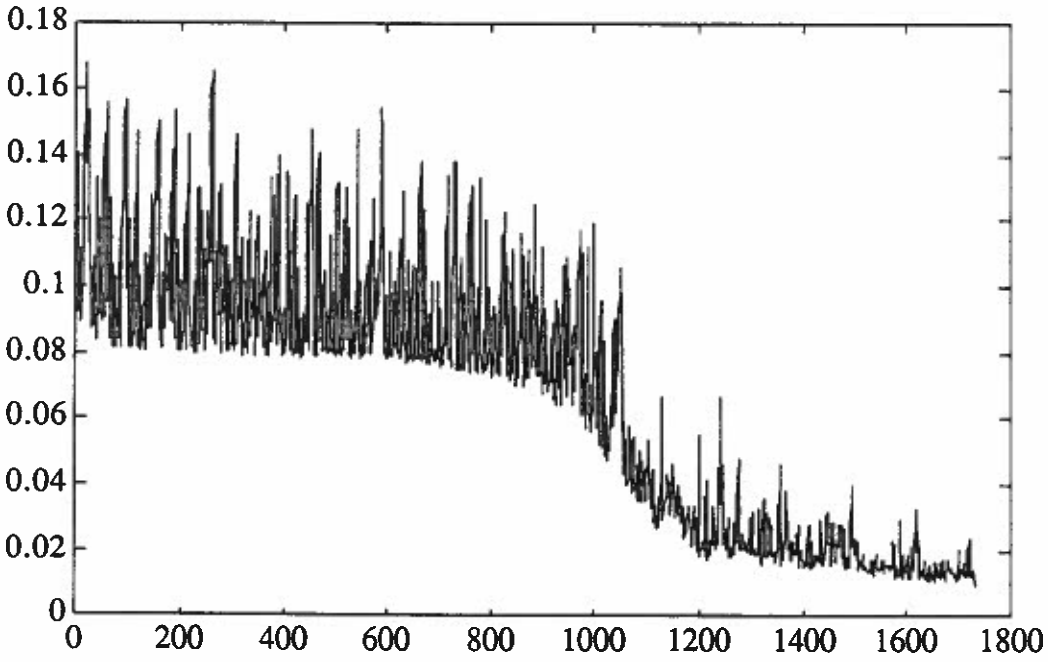

Figure 2 Plot of $\hat{F}$ for the back-propagation algorithm.

step 7) is able to correctly approximate the desired mapping of the training set. For #1 = 2 and #1 = 3, the Quadratic Optimization Algorithm was unable to find values for the weights such that the desired mapping was achieved, while the back-propagation algorithm was able to successfully find such weights.

To illustrate some of the observations that are made at the end of Section 5, the weights and the outputs of a neural network found via the Quadratic Optimization Algorithm are presented. For the neural network with 5 hidden layer neurons, the output layer weights found using the Quadratic Optimization Algorithm for 1 iteration are

$$w^2 = \begin{matrix} 10.7469 \\ 53.1767 \\ 49.9710 \\ 16.5357 \\ 21.8128 \\ 0.5822 \end{matrix}$$

and those found using the back-propagation algorithm for 1729 iterations are

$$w^2 = \begin{matrix} -0.3481 \\ 0.3337 \\ 0.0400 \\ 0.0223 \\ 0.0016 \\ 0.1508 \end{matrix}$$

As described previously, the weights for the output layer may be large, and they are for this case. However, in the simulations using the training sets of the other examples, these values may be several orders of magnitude larger than the ones for this extended XOR example.

As another comparison for the neural network with 5 hidden layer neurons, the outputs of the neural network found using the Quadratic Optimization Algorithm for 1 iteration are

$$y = \begin{matrix} 0.0079 \\ -0.0256 \\ -0.0554 \\ 0.0170 \\ 0.1238 \\ -0.1056 \\ -0.0906 \\ 0.1233 \end{matrix}$$

and those found using the back-propagation algorithm for 1729 iterations are

$$y = \begin{matrix} 0.0531 \\ -0.0802 \\ -0.0789 \\ 0.0514 \\ 0.1526 \\ -0.0809 \\ -0.0783 \\ 0.1165 \end{matrix}$$

where the magnitude of the desired output is 0.1. Clearly, the neural network trained with the Quadratic Optimization Algorithm did not achieve the desired neural network outputs but did achieve the desired mapping for the classification training set. For this reason, it is recommended that the Quadratic Optimization Algorithm be used for classification problems and not for general function approximation problems. However, even though the exact desired outputs may not be achieved using the Quadratic Optimization Algorithm, the resulting neural network does have desirable generalization properties for classification training sets as illustrated in the following examples.

Example 6.3:

In a square of size [0, 8] x [0, 8], consider a circle of radius 2 centered at (4, 4). Let the input patterns be points inside the square. If the input pattern lies inside the circle, the corresponding desired output is 1, and if the input pattern lies outside the circle, the corresponding desired output is -1. Thus, $\mathbf{U} \in \mathbb{R}^{3xp}$ and $\mathbf{d} \in \mathbb{R}^{px1}$. The training patterns are taken as points evenly spaced over the square; a new pattern occurs every 1.0 steps in a direction parallel to an axis for a total of p = 81 training patterns with 13 inside the circle and 68 outside the circle such that $\mathbf{U} \in \mathbb{R}^{3x81}$ and $\mathbf{d} \in \mathbb{R}^{81x1}$. In Figure 3, the training patterns are indicated with the appropriate desired output, as well as the circle for reference. A two-layer neural network is provided with #1 = 15. The hyperbolic tangent function is used as the nonlinearity for the hidden layer's neurons, and the signum function is used as the nonlinearity for the output layer's neurons. The neural network is trained with the Quadratic Optimization Algorithm. The algorithm is implemented in MATLAB on a Sun Sparc Station using a program that is not optimized. Steps 5) and 7) of the algorithm are solved using the psuedo-inverse function call for MATLAB. Since the signum is the output layer's nonlinearity, the vector v^2 is chosen such that $v^2(j) = 0.1(d(j))$ for $1 \leq j \leq 81$. The weights for the neural network are chosen at random in the interval $[-\frac{1}{15}, \frac{1}{15}]$ such that $\hat{F}(0) = 0.9711$. After applying the Quadratic Optimization Algorithm for one iteration requiring 835467 floating point operations, $\hat{F}(1) = 0.1536$ and the training set is correctly classified. To test the generalization ability of the result, the neural network is probed with inputs occurring at a 0.5 interval, most of which were not used in the training set. The resulting output is plotted in Figure 4. Clearly, the neural network has generalized well over the input space.

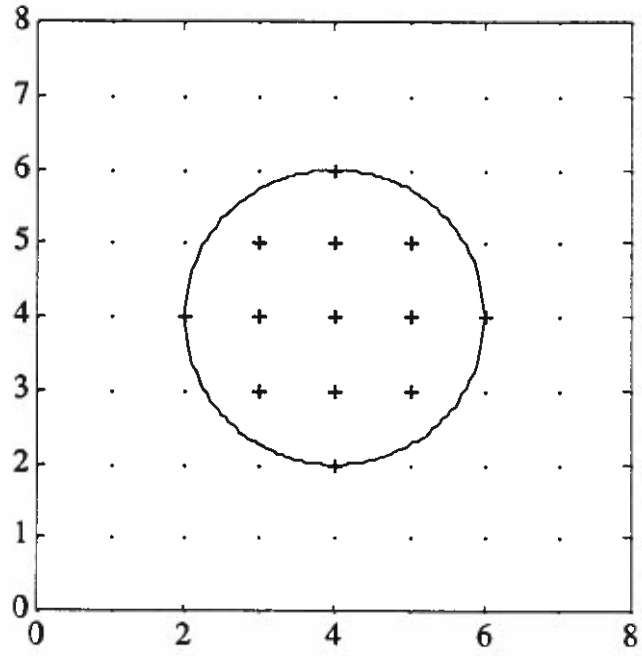

Figure 3 Training set for Example 6.3.

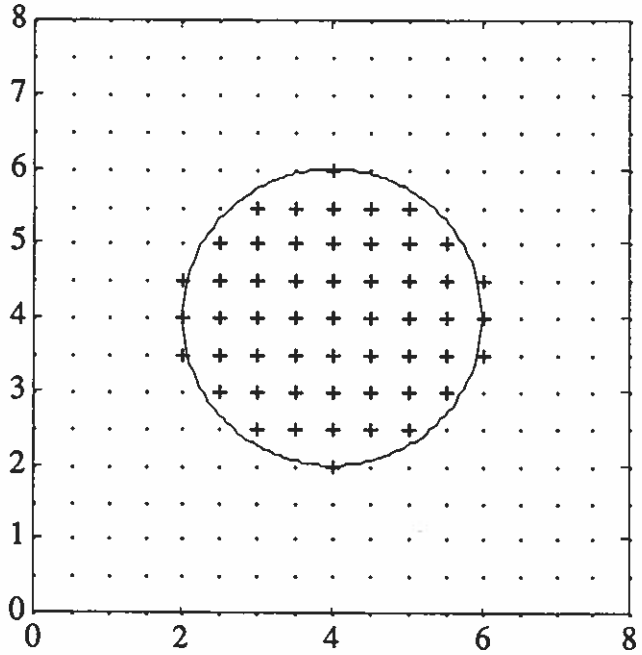

Figure 4 Testing the trained neural network of Example 6.3.

M. A. Sartori and P. J. Antsaklis, "Neural Network Training via Quadratic Optimization," Technical Report #90-05-01, Dept. of Electrical and Computer Engineering, University of Notre Dame, May 1990. Revised April 1991.

Example 6.4:

The same circular area as in Example 6.3 except with a smaller number of training patterns is used for this example. The training patterns occur at 2.0 intervals across the square for a total of $p = 25$ training patterns with 5 inside the circle and 20 outside the circle and are depicted in Figure 5. Thus, $\mathbf{U} \in \mathbb{R}^{3x25}$ and $\mathbf{d} \in \mathbb{R}^{25x1}$. A two-layer neural network is provided with #1 = 17. The hyperbolic tangent function is used as the nonlinearity for the hidden layer's neurons, and the signum function is used as the nonlinearity for the output layer's neurons. The neural network is trained with the Quadratic Optimization Algorithm, which is implemented in MATLAB on a Sun as described in Example 6.3. Since the signum function is the output layer's nonlinearity, the vector $\mathbf{v}^2$ is chosen such that $v^2(j) = 0.1(d(j))$ for $1 \leq j \leq 25$. The weights for the neural network are chosen at random in the interval $[-\frac{1}{17}, \frac{1}{17}]$ such that $\hat{F}(0) = 0.1888$. After applying the Quadratic Optimization Algorithm for one iteration requiring 213638 floating point operations, $\hat{F}(1) = 0.0332$ and the training set is correctly classified. To test the generalization ability of the result, the neural network is probed with inputs occurring at a 0.5 interval. The resulting output is plotted in Figure 6. Once again, the trained neural network generalizes well over the input space.

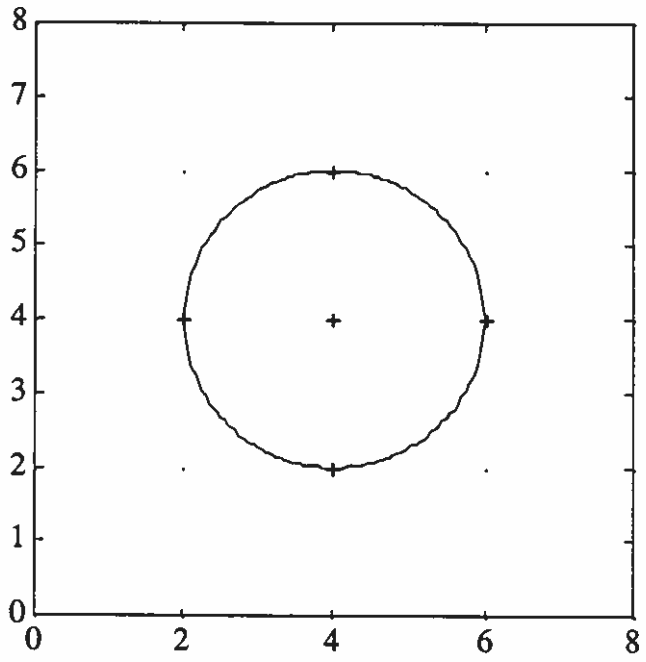

Figure 5 Training set for Example 6.4.

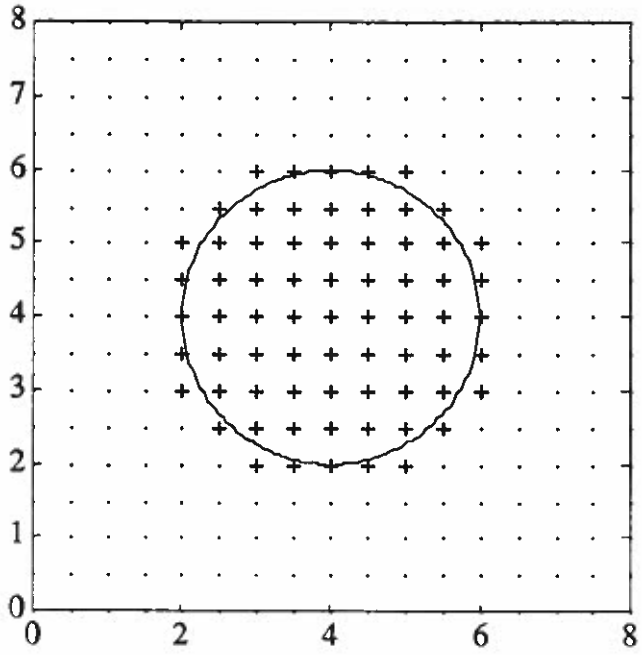

Figure 6 Testing the trained neural network of Example 6.4.

Example 6.5:

The same circular area as in Examples 6.3 and 6.4 except with randomly placed patterns is used for this example. The 289 training patterns are chosen at random with a uniform distribution over the [0, 8] x [0, 8] square region and are depicted in Figure 7. Thus, $\mathbf{U} \in \mathbb{R}^{3x289}$ and $\mathbf{d} \in \mathbb{R}^{289x1}$. A two-layer neural network is provided with #1 = 30. The hyperbolic tangent function is used as the nonlinearity for the hidden layer's neurons, and the signum function is used as the nonlinearity for the output layer's neurons. The neural network is trained with the Quadratic Optimization Algorithm, which is implemented in MATLAB on a Sun as described in Example 6.3. Since the signum function is the output layer's nonlinearity, the vector $\mathbf{v}^2$ is chosen such that $v^2(j) = 0.1(d(j))$ for $1 \leq j \leq 289$. The weights for the neural network are chosen at random in the interval $[-\frac{1}{30}, \frac{1}{30}]$ such that $\hat{F}(0) = 3.7221$. After applying the Quadratic Optimization Algorithm for one iteration requiring 14725850 floating point operations, $\hat{F}(1) = 0.3472$ and the training set is almost arbitrarily correctly classified. To test the generalization ability of the result, the neural network is probed with 1089 randomly chosen patterns. The resulting output is displayed in Figure 8. Once again, the trained neural network clearly generalizes well over the input space.

M. A. Sartori and P. J. Antsaklis, "Neural Network Training via Quadratic Optimization," Technical Report #90-05-01, Dept. of Electrical and Computer Engineering, University of Notre Dame, May 1990. Revised April 1991.

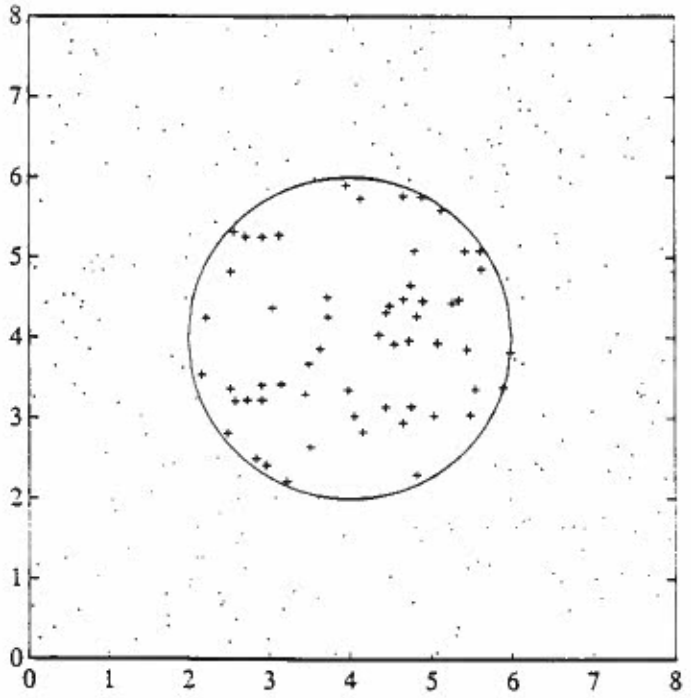

Figure 7 Training set for Example 6.5.

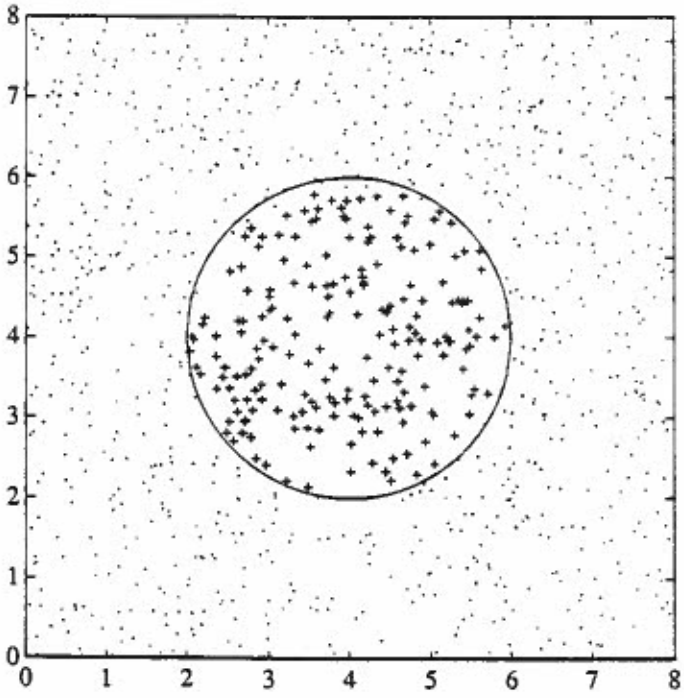

Figure 8 Testing the trained neural network of Example 6.5.

Example 6.6:

A variation of the previous three examples is presented here. Instead of a single circle in the [0, 8] x [0, 8] square, two circles centered at (2, 2) and (6, 6) and each with a radius of 1 are placed in the square such that an XOR-type classification problem results. Shown in Figure 9, the training patterns are placed at 1.0 intervals across the square, and $\mathbf{U} \in \mathbb{R}^{3x81}$ and $\mathbf{d} \in \mathbb{R}^{81x1}$. A two-layer neural network is provided with #1 = 30. Once again, the hyperbolic tangent function is used as the nonlinearity for the hidden layer's neurons, and the signum function is the used as the nonlinearity for the output layer's neurons. The neural network is trained with the Quadratic Optimization Algorithm, which is implemented in MATLAB on a Sun as described in Example 6.3. The vector $\mathbf{v}^2$ is chosen such that $v^2(j) = 0.1(d(j))$ for $1 \leq j \leq 81$. The weights for the neural network are chosen at random in the interval $[-\frac{1}{30}, \frac{1}{30}]$ such that $\hat{F}(0) = 0.7050$. After applying the Quadratic Optimization Algorithm for one iteration requiring 1943422 floating point operations, $\hat{F}(1) = 0.1040$ and the training set is correctly classified. To test the generalization ability of the result, the neural network is probed with inputs occurring at a 0.5 interval. The resulting output is plotted in Figure 10, and the trained neural network once again generalizes well over the input space for a classification problem.

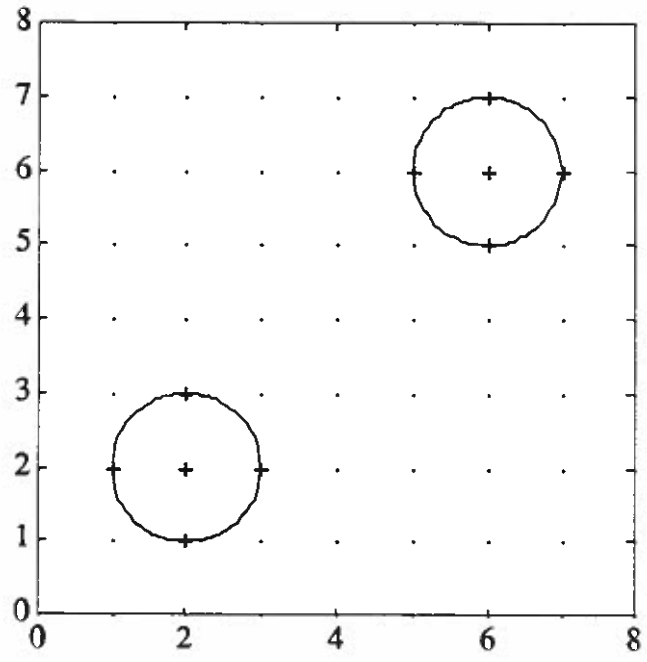

Figure 9 Training set for Example 6.6.

M. A. Sartori and P. J. Antsaklis, "Neural Network Training via Quadratic Optimization," Technical Report #90-05-01, Dept. of Electrical and Computer Engineering, University of Notre Dame, May 1990. Revised April 1991.

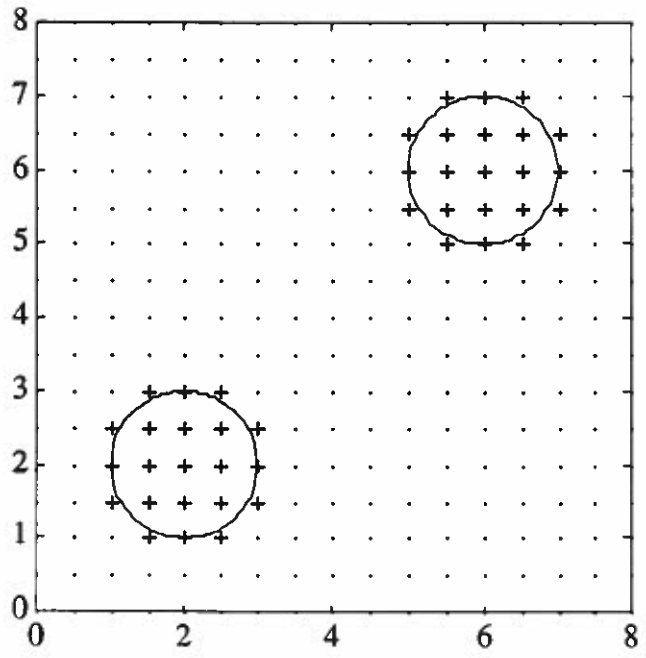

Figure 10 Testing the trained neural network of Example 6.6.

Example 6.7:

The reproduction of a sinusoid with amplitude 0.5 over one period is desired for this example. Sampling (0.5)sin(k) for k = 0, $2\pi/8$, ..., 2π, the input matrix $\mathbf{U} \in \mathbb{R}^{2x9}$ and the desired vector $\mathbf{d} \in \mathbb{R}^{9x1}$ are formed. The training set is shown in Figure 11. A two-layer neural network is provided with #1 = 5. The hyperbolic tangent function is used as the nonlinearity for the hidden layer neurons, and the linear function is the nonlinearity for the output layer neurons. The neural network is trained with the Quadratic Optimization Algorithm, which is implemented in MATLAB on a Macintosh SE as described in Example 6.2. The vector $\mathbf{v}^2$ is chosen such that $v^2(j) = d(j)$ for $1 \leq j \leq 9$. The weights for the neural network are chosen at random in the interval $[-\frac{1}{5}, \frac{1}{5}]$ such that $\hat{F}(0) = 1.3388$. After applying the Quadratic Optimization Algorithm for one iteration requiring 10720 floating point operations, $\hat{F}(1) = 9.4942e\text{-}06$ and the training set is correctly classified. To test the generalization ability of the result, the neural network is probed with different input values for k, where now k = 0, $2\pi/25$, ..., 2π. The resulting output is plotted in Figure 12 along with the sinusoid, and the trained neural network generalizes well over the input space even though the desired mapping is not a classification problem.

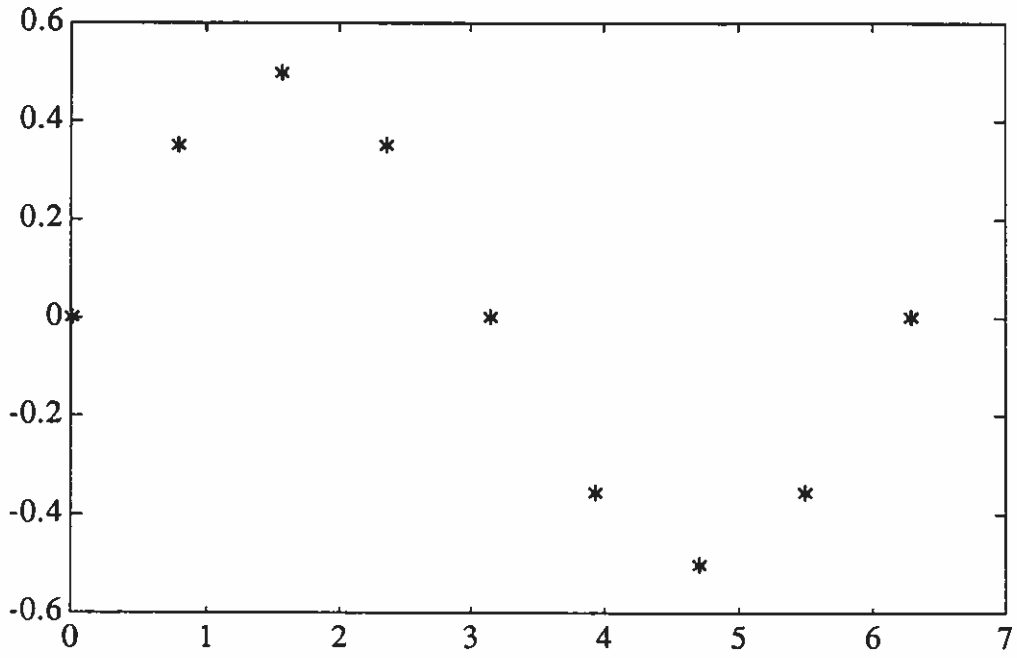

Figure 11 Training set for Example 6.7.

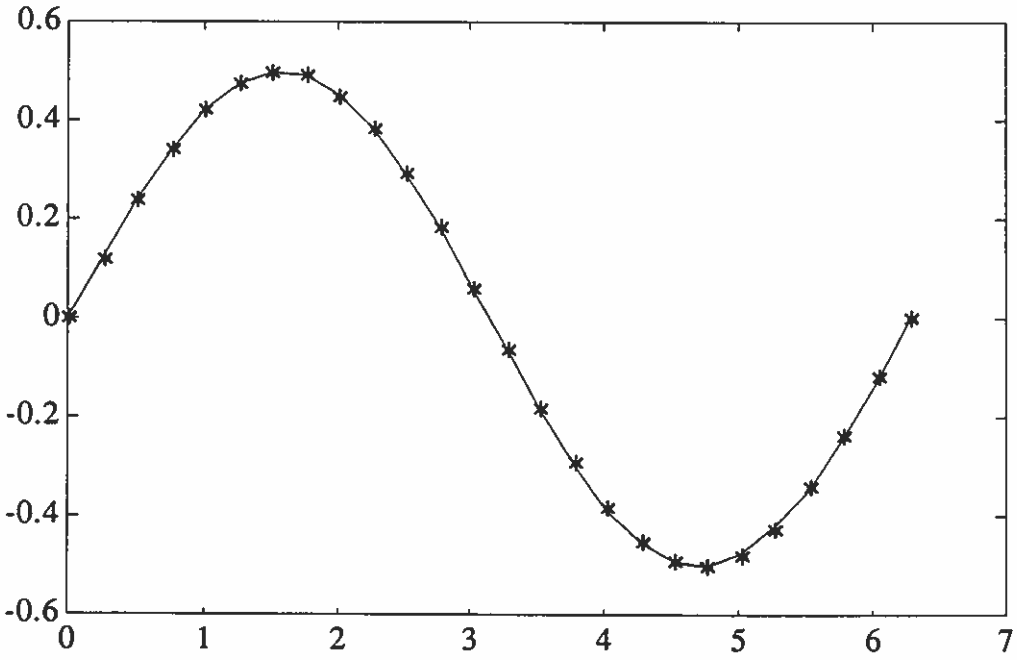

Figure 12 Testing the trained neural network of Example 6.7.

7 CONCLUDING REMARKS

A new method based on the minimization of a quadratic function is presented for the training of a single neuron, a single-layer neural network, and a multi-layer neural network. An examination and error analysis is provided when the new quadratic optimization procedure is applied to a single neuron. These results can be immediately applied to the single-layer neural network case, but not for the multi-layer neural network case since there do not exist known desired outputs for the neural network's hidden layers. By using the concept of back-propagating the output error to the hidden layers, desired outputs are approximated for the hidden layers, and the results for the single-layer neural network are applied to each of the hidden layers. The training of a multi-layer neural network via the described Quadratic Optimization Algorithm tends to work best for classification problems and tends to achieve good results in a single iteration.

The results reported in this paper also appear in [Sartori91].

8 REFERENCES

[Bazaraa79] Bazaraa M.S., Shetty C.M., *Nonlinear Programming: Theory and Applications*, Wiley, New York, 1979.

[Bertsekas89] Bertsekas D.P., Tsitsiklis J.N., *Parallel and Distributed Computation: Numerical Methods*, Prentice Hall, 1989.

[Gill81] Gill P.E., Murray W., Wright M.H., *Practical Optimization*, Academic Press, New York, 1981.

[Rosenblatt62] Rosenblatt F., *Principles of Neurodynamics*, Spartan, New York, 1962.

[Rumelhart86] Rumelhart D.E., Hinton G.E., Williams R.J., "Learning Internal Representations by Error Propagation," in Rumelhart D.E., McClelland J.L., eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol. 1: Foundation*, pp. 318-362, MIT Press, 1986.

[Sartori91] Sartori M.A., *Feedforward Neural Networks and Their Application in the Higher Level Control of Systems*, Ph.D. Dissertation, Department of Electrical Engineering, University of Notre Dame, April 1991.

[Shynk90] Shynk J.J., "Performance Surfaces of a Single-Layer Perceptron," *IEEE Transactions on Neural Networks*, vol. 1, no. 3, pp. 268-277, Sept. 1990.

[Widrow60] Widrow B., Hoff Jr. M.E., "Adaptive Switching Circuits," in *IRE WESCON Convention Record*, pt. 4, pp. 96-104, Sept. 1960.