Training feed-forward neural networks using conjugate gradients

James L. Blue and Patrick J. Grother

National Institute of Standards and Technology
Gaithersburg, MD 20899

Abstract

Neural networks for optical character recognition are still being trained using back propagation, even though conjugate gradient methods have been shown to be much faster. Most multilayer perceptron network training results in the literature are obtained for small and unrealistic problems or from data sets that are proprietary and not available for comparison testing.

We present results on a large realistic pattern set containing 2000 training and 1434 testing exemplars. Each pattern is composed of 32 Gabor coefficients obtained from a 32 by 32 pixel binary image of a hand-written digit segmented from the NIST Handwriting Image Data Base. These sets are believed to have approximately 1% segmentation errors. Comparative results for Møller's scaled conjugate gradient method and for standard back propagation are presented for runs on a serial scientific workstation and a highly parallel computer.

Typical training on a network with 32 inputs, 32 hidden nodes, and 10 output nodes gives a 98% to 99% recognition for the training set and 95% for the test set. Training with conjugate gradients requires fewer than 200 iterations; times are about 20 to 40 minutes on a scientific workstation and 6 minutes on the highly parallel computer. Testing (classification) is done at the rate of 600 and 1600 patterns per second on the scientific workstation and on the highly parallel computer respectively.

These results suggest that commercial hand-written character recognition systems with great economic potential are feasible.

1. Introduction

The success of neural network technology in pattern recognition is exemplified by its application to the specific field of optical character recognition. The commercial value of automatic document processing systems has attracted much research and investment in the relevant areas of image preprocessing, segmentation, recognition and understanding. Identification of machine printed digits is a solved problem but that of unconstrained handwritten characters is less tractable. The self organizing architectures offer a neural technology to address this problem. In particular Vogl's[1] and Grossberg's ART[2] methods construct distributed representations of the input patterns. However, arguably the most common neural network technology, namely the multilayer perceptron architecture, has repeatedly been applied to this problem using a variety of representative input features matched with desired outputs. Notable performance has been obtained but, with superior recognition remaining a goal, numerous methods of improving neural technologies are on going. The results detailed in this paper show very significant improvements in the method of training feed forward networks are readily obtainable. Benefits extend to perceptron applications beyond character recognition such as speech synthesis and control.

Backpropagation (BP) has been used for some years[3] to train feed-forward perceptrons. Mathematically, "training" means minimizing an error function, the sum of the squares of the errors in the outputs. While useful for training networks, BP has the disadvantages that convergence is slow[4] and that there are, in the usual implementation[3] two adjustable parameters, η and α, that have to be determined. A third disadvantage, that convergence is often to a local minimum of the error function rather than a global minimum, is inherent in the problem and not restricted to BP.

Since BP corresponds (approximately) to using a steepest-descent method for minimizing the error function, and steepest-descent methods are known[5,6] to converge slowly when near a minimum, slow convergence is not surprising.

Conjugate gradient (CG) methods have been used for many years[7,8] for minimizing functions, and have recently[9] been discovered by the neural network community. The usual CG methods require a line search or its equivalent. Møller[10] has introduced a scaled conjugate gradient (SCG) method; instead of a line search, he uses an estimate of the second derivative along the search direction to find the approximate minimum.

Many BP implementations for feed-forward networks are available, but CG and SCG implementations have not been available. We have produced simple and easy-to use SCG and BP programs that share a common driver program. These programs are available from the authors by electronic mail.[1]

We also present a large and realistic test set of data for use with the SCG and BP programs. The data are coefficients from a collection of handwritten digits. The data set will be made available on the neuroprose archives.

1.1. The problem

We assume a standard multilayer perceptron network, fully-connected, with $N^{(0)}$ inputs, $N^{(1)}$ hidden neurons, and $N^{(2)}$ output neurons. Define $o_{ip}^{(k)}$ to be the activation of the i^{th} element of the k^{th} layer of the network, due to the p^{th} pattern of inputs, with $o_{ip}^{(0)}$ the input value for neuron i. Define the weight matrix interconnecting the elements of neuron layer $m-1$ to layer m to be $w_{ij}^{(m)}$, and let f be some nonlinear squashing function, such as the standard sigmoid function, $f(x) = 1/[1+\exp(-x)]$. The input pattern vectors propagate forward through successive layers of neurons according to

$$x_{hp}^{(1)} = \sum_{i=0}^{N^{(0)}} w_{hi}^{(1)} o_{ip}^{(0)}; \qquad o_{hp}^{(1)} = f\left(x_{hp}^{(1)}\right);$$
$$x_{jp}^{(2)} = \sum_{h=0}^{N^{(1)}} w_{jh}^{(2)} o_{hp}^{(1)}; \qquad o_{jp}^{(2)} = f\left(x_{jp}^{(2)}\right); \tag{1}$$

where $o_{0p}^{(0)} \equiv 1$ and $o_{0p}^{(0)} \equiv 1$, so that $w_{0h}^{(1)}$ and $w_{0j}^{(2)}$ are biases for the squashing functions.

Pattern p has a desired output activation, or target value, of v_{jp} at output neuron j. For character classification, each pattern has exactly one v_{jp} equal to one and the rest equal to zero. Assume a set of $N^{(P)}$ patterns, a set to be used for "training" the network. One measure of the goodness of a set of weights is

$$\mathcal{E} = \frac{1}{2N^{(2)}N^{(P)}} \sum_{p=1}^{N^{(P)}} \sum_{j=1}^{N^{(2)}} \left[v_{jp} - o_{jp}^{(2)}\right]^2; \tag{2}$$

the root-mean-square (RMS) error, $\sqrt{\mathcal{E}}$, is a measure of the output activation error averaged over all patterns and all output neurons. Minimizing $\mathcal{E}$ provides one way of determining the weight matrices $w_{hi}^{(1)}$ and $w_{jh}^{(2)}$; it is a substitute for what is really desired, to determine $w_{hi}^{(1)}$ and $w_{jh}^{(2)}$ so as to do the best possible character identification over some large and unknown set of "testing" input patterns.

The negative gradients of $\mathcal{E}$ with respect to the weights are

$$g_{jh}^{(2)} = -\frac{\partial \mathcal{E}}{\partial w_{jh}^{(2)}} \quad \text{and} \quad g_{hi}^{(1)} = -\frac{\partial \mathcal{E}}{\partial w_{hi}^{(1)}}. \tag{3}$$

[1]Send electronic mail to **jlb@azure.cam.nist.gov**; include your name, institution, address, and electronic mail address.

Define auxiliary variables $\delta_{hp}^{(1)}$ and $\delta_{jp}^{(2)}$:

$$\delta_{jp}^{(2)} = \left(v_{jp} - o_{jp}^{(2)}\right) f'(x_{jp}^{(2)}) \quad \text{and} \quad \delta_{hp}^{(1)} = \sum_{j=0}^{N^{(2)}} \delta_{jp}^{(2)} w_{jh}^{(2)} f'(x_{hp}^{(1)}), \tag{4}$$

where $f'(x)$ is the derivative of f; for the usual sigmoid function, $f'(x) = f(x)[1-f(x)]$. Then the negative gradients are

$$g_{jh}^{(2)} = \frac{1}{N^{(2)}N^{(p)}} \sum_{p=1}^{N^{(p)}} \delta_{jp}^{(2)} o_{hp}^{(1)} \quad \text{and} \quad g_{hi}^{(1)} = \frac{1}{N^{(2)}N^{(p)}} \sum_{p=1}^{N^{(p)}} \delta_{hp}^{(1)} o_{ip}^{(0)}. \tag{5}$$

To simplify the notation in later sections, let $\mathbf{w}$ be a vector of length $N^{(w)} = [N^{(0)}+1]N^{(1)} + [N^{(1)}+1]N^{(2)}$, formed by concatenating all the $w_{hi}^{(1)}$ and $w_{jh}^{(2)}$ in some order. Similarly, $\mathbf{g}$ is the vector of negative gradients of $\mathcal{E}$ with respect to $\mathbf{w}$, formed by concatenating all the $g_{hi}^{(1)}$ and $g_{jh}^{(2)}$ in the same order.

The standard "network training" problem then reduces to finding a set of weights, $\mathbf{w}$, to minimize the mean-square activation error, $\mathcal{E}$. An alternative minimization problem is a regularized version,[11] sometimes known as the Levenberg-Marquardt method: minimize a linear combination of the mean-square activation error and the mean-square weight:

$$\mathcal{E}(\mathbf{w};\mu) = \mathcal{E} + \frac{\mu}{N^{(w)}} \left(\sum_{i=0}^{N^{(0)}} \sum_{h=1}^{N^{(1)}} \left[w_{hi}^{(1)}\right]^2 + \sum_{h=0}^{N^{(1)}} \sum_{j=1}^{N^{(2)}} \left[w_{jh}^{(2)}\right]^2 \right). \tag{6}$$

There is an additional parameter, μ, which must be chosen somehow. Since $\mathcal{E} = \mathcal{E}(\mathbf{w};0)$, the remainder of this paper will use $\mathcal{E}(\mathbf{w};\mu)$. The negative gradients become

$$g_{jh}^{(2)} = \frac{1}{N^{(2)}N^{(p)}} \sum_{p=1}^{N^{(p)}} \delta_{jp}^{(2)} o_{hp}^{(1)} - \frac{\mu}{N^{(w)}} w_{jh}^{(2)}$$

$$g_{hi}^{(1)} = \frac{1}{N^{(2)}N^{(p)}} \sum_{p=1}^{N^{(p)}} \delta_{hp}^{(1)} o_{ip}^{(0)} - \frac{\mu}{N^{(w)}} w_{hi}^{(1)}. \tag{7}$$

Minimization of a function of many variables is a mature field;[12] there are numerous algorithms and more numerous variations of them. All are iterative, generating a sequence of sets of weights. For neural network applications in character recognition, where typical values of $N^{(w)}$ might range from a few hundred to many thousand, computationally feasible algorithms can use the gradient vector but not the Hessian matrix (the matrix of second derivatives of the function). The typical algorithm has the structure

1. Choose an initial vector $\mathbf{w}_0$ and set $k = 0$. In the absence of a better idea, random starting values are used.
2. Calculate the function, $\mathcal{E}(\mathbf{w}_k;\mu)$, and its negative gradient, $\mathbf{g}_k$.
3. Calculate a step, $\Delta\mathbf{w}_k$. Let $\mathbf{w}_{k+1} = \mathbf{w}_k + \Delta\mathbf{w}_k$.
4. Calculate the function, $\mathcal{E}(\mathbf{w}_{k+1};\mu)$, and its negative gradient, $\mathbf{g}_{k+1}$.
5. If the $\mathbf{w}_{k+1}$ is unsatisfactory, make some adjustments, discard $\mathbf{w}_{k+1}$ and go to 3. Otherwise, increase k by 1.
6. If the iteration has converged, or if too many iterations have been done, or if progress seems hopeless, quit.
7. Go to 3.

This generalized procedure, with suitable elaborations of steps 3, 5, and 6, always succeeds in finding *a* local minimum of the function. There is no practical way to find the *absolute* minimum, and $\mathcal{E}(\mathbf{w};\mu)$ has many local minima. Essentially all the computer time is spent in step 4.

1.2. Backpropagation minimization of $\mathcal{E}(\mathbf{w};\mu)$

For a given $\mathbf{w}$, $\mathbf{g}$ is the direction of *steepest descent* for $\mathcal{E}(\mathbf{w};\mu)$, the direction in which $\mathcal{E}(\mathbf{w};\mu)$ decreases most rapidly. Many variations are possible. One possibility is

$$\Delta\mathbf{w}_k = \eta_k \mathbf{g}_k \tag{8}$$

with η_k determined by a line search, approximately minimizing the function with respect to η_k. The standard backpropagation method[3] uses

$$\Delta\mathbf{w}_k = \eta \mathbf{g}_k + \alpha\Delta\mathbf{w}_{k-1} \tag{9}$$

with fixed parameters η and α, not determined by the algorithm. For large problems, (9) works well for a few iterations, then slows down drastically.[4-6] An example is given in Section 4.

1.3. Conjugate gradient minimization of $\mathcal{E}(\mathbf{w};\mu)$

For many years, conjugate gradient algorithms have been used to minimize functions[7,8] and good Fortran programs have been available.[13] More recently, the neural network community has discovered conjugate gradients.[9,10] The step used is

$$\Delta\mathbf{w}_k = \alpha_k(\mathbf{g}_k + \beta_k\Delta\mathbf{w}_{k-1}) \tag{10}$$

where β_k is calculated by the algorithm to make $\Delta\mathbf{w}_k$ and $\Delta\mathbf{w}_{k-1}$ *conjugate*, or orthogonal in a generalized meaning of the word. The factor α_k is often determined by some kind of line search, using about three function calls.

Møller[10] uses a (temporary) small value for α_k, does a function evaluation in order to approximate the second derivative in the search direction, then uses the approximate second derivative to select a final α_k: this uses two function calls. (If the new weights result in an increased error, they are rejected and some parameters are adjusted until the new weights reduce the error. In this case, a successful iteration takes more than two function calls.) Møller calls his method Scaled Conjugate Gradients (SCG). Our experiments on character recognition problems showed SCG to be preferable to the program of Shanno and Phua.[13]

2. Programs

Fortran programs to train neural networks using BP and SCG have been written: the BP and SCG programs differ only in the subprogram called to minimize $\mathcal{E}(\mathbf{w};\mu)$. The implementations for serial computers are reasonably straightforward; the implementation for parallel computers will be discussed in Section 2.1.

Deciding when to stop an iterative procedure is not straightforward. In principle, stopping is simple: stop when a local minimum has been reached. At a local minimum, $\mathbf{g} = \mathbf{0}$. (The same is true at at a saddle point or a local maximum, but checking for these requires computing the Hessian matrix, which is impractical.) However, in finite-precision arithmetic $\mathbf{g} = \mathbf{0}$ is attained only in unusual circumstances. The programs have five stopping criteria built in:

1.	$k >= C_1$	(Too many iterations)
2.	$\mathcal{E}(\mathbf{w}_k;\mu) <= C_2$	(Achieved error goal)
3.	$\lvert\mathbf{g}_k\rvert <= C_3\lvert\mathbf{w}_k\rvert$	(Achieved gradient goal)
4.	$\mathcal{E}(\mathbf{w}_k;\mu) >= (1 - C_4)E_{k-K}$	(Error decreasing too slowly)
5.	$R_k >= R_{k-K} - C_5$	(Right classifications decreasing too slowly)

where C_1, C_2, C_3, C_4, C_5, and K are constants supplied by the user. (Criteria 4 and 5 are checked only every K iterations.) The results obtained depend significantly on which criterion applies in the particular run, which in turn depends on the constants. Some examples are given in Section 4.

2.1. Parallel implementation

The efficient parallel implementation of neural networks in hardware and software is an active area of current research. The motivation is pragmatic; faster training algorithms and their implementation allow more complicated larger networks to be evaluated. The parallelism inherent in artifical neural nets is well known. In particular the multilayer perceptrons trained by BP and SCG are both readily made parallel. Their implementation is particularly suited to massively parallel fine grained SIMD architectures. Machines of this type, such as the AMT DAP or the Loral Industries MPP[2], are typified by tightly coupled processors connected by high bandwidth bus.

2.1.1. Forward propagation

The multilayer perceptron networks require that pattern vectors to be propagated forward through successive layers of neurons. Define activation matrices $\mathbf{\Phi}^{(1)}$ and $\mathbf{\Phi}^{(2)}$ whose columns are the activation vectors for each pattern. The parallel equations corresponding to 1 are

$$\mathbf{\Phi}^{(1)} = f\left(\mathbf{W}^{(1)}\mathbf{\Phi}^{(0)}\right) \quad \text{and} \quad \mathbf{\Phi}^{(2)} = f\left(\mathbf{W}^{(2)}\mathbf{\Phi}^{(1)}\right). \tag{11}$$

The activations of neurons within a layer are independent and their parallel calculation is thus suggested. The problem is then one of matrix multiplication.

In the serial computer the choice of the matrix multiplication algorithm used is largely irrelevant, since all operations are scalar. Although the literature on parallel matrix multiplication is vast the *outer product*[14] method on the array processor[3] is found to be superior if the matrices are much larger than the number of processors. The following pseudocode multiplies two matrices, m1 and m2, obtaining the product m3.

```
real m1(L, M)                # L rows x M columns matrix
real m2(M, N)                # M rows x N columns matrix
real m3(L, N)                # L rows x N columns product

real acol(L), arow(N)        # rows and columns of m1 and m2
real row_replicas(L, N)      # arow is broadcast down this matrix
real col_replicas(L, N)      # acol is broadcast across this matrix

m3 = 0.0
do i = 1, M
{
  acol = m1(,i)                              # col i of m1
  arow = m2(i,)                              # row i of m2
  row_replicas = row_broadcast(arow, L)      # all rows the same
  col_replicas = col_broadcast(acol, N)      # all cols the same
  m3  = m3 + row_replicas * col_replicas
}
```

The expensive operations here are the parallel multiplication and the subsequent parallel accumulation. The more naïve *inner product* algorithm computes dot products thus:

[2]Certain commercial equipment is identified in order to adequately specify or describe the subject matter of this work. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the equipment identified is necessarily the best available for the purpose.

[3]An array processor is considered here to be a finite two dimensional rectangular array of SIMD processing elements. Operations on matrices larger than this array are looped transparently by the machine.

```
real m1(L, M)                 # L rows x M columns matrix
real m2(M, N)                 # M rows x N columns matrix
real m3(L, N)                 # L rows x N columns result

real arow(N)                  # rows of m1 and m2
real col_replicas(L, N)       # acol is broadcast across this matrix

do i = 1, N
{
  arow = m1(i,)
  col_replicas = col_broadcast(arow, N)          # all cols the same
  m3(,i) = sum_over_rows(m2 * col_replicas)      # sum all columns
}
```

The parallel product is inexpensive: the inefficiency of the method is determined by the speed of interprocessor communication necessary for the term collection in the "sum over rows." The outer product algorithm is not dependent on this bandwidth and is therefore faster.

2.1.2. Weight Modification

Forward propagation is necessary in training and in classification. During training the weight update is dependent on the error gradient. The error, E, is given as the difference between the target output and the actual output thus:

$$\mathbf{E} = \mathbf{\Psi} - \mathbf{\Phi}^{(2)} \tag{12}$$

The output layer error is weighted by the first deriviative of the squashed output activation

$$\mathbf{\Delta}^{(2)} = \mathbf{E} \otimes f'\left(\mathbf{\Phi}^{(2)}\right) \tag{13}$$

where $\otimes$ denotes parallel component-by-component multiplication. The hidden layer error is obtained by passing the output deltas back through the final weight layer.

$$\mathbf{\Delta}^{(1)} = \left(\mathbf{W}^{(2)}\mathbf{\Delta}^{(2)}\right) \otimes f'\left(\mathbf{\Phi}^{(1)}\right) \tag{14}$$

The negative gradients of the error with respect to the individual weights are

$$\mathbf{g}^{(1)} = \mathbf{\Phi}^{(0)}\mathbf{\Delta}^{(1)^T} \quad \text{and} \quad \mathbf{g}^{(2)} = \mathbf{\Phi}^{(1)}\mathbf{\Delta}^{(2)^T} \tag{15}$$

The weights are independent and therefore the update operations are homogenous and purely parallel. The weight update time is negligible compared to doing the forward propagation and gradient calculation.

3. The handwritten characters database

The data set used for the examples in this paper is known as FL3; it contains features extracted from handprinted digits. There are 2000 patterns in the training set and 1434 in the testing set.

The inital character images were obtained by isolating and segmenting full page images from the NIST Handprinted Characters Database;[15] segmentation is described in Wilkinson.[16] The resulting character images were centered in a 32×32 array of pixels. Each character image then had a shear transformation a Gabor feature extraction, reducing each 1024-pixel binary image to a 32-feature pattern.

3.1. Shear Transformation

The purpose of the shear transform is to remove the slant from the character images. Pixel histograms are calculated for the top and bottom few rows of the image. The centers of these two distributions determine a virtual line between them, which is used to construct the edges of a parallelogram around the character. Each row of the image is shifted horizontally to bring the edges of the parallelogram into vertical alignment.

3.2. Gabor Feature Extraction

In classifying images, each 32 pixels by 32 pixels in size, to decide which character is represented in each image, A neural network would need 1024 input values, one for each pixel, if the image pixel values were fed directly to the network. This network is larger than necessary. To get a smaller network, the essence of each image must somehow be distilled into a smaller group of numbers, each describing some *feature*, with the value of each feature used as one of the input values.

Various features are possible. Recently the Gabor functions[17] have been proposed as a model of mammalian visual receptive fields[18] and as such are a biologically based method of extracting salient features from characters. Such features have been used with considerable success in machine print OCR[19] and handprint OCR.[20]

3.2.1. Gabor Filtering

Gabor functions come in even and odd variants:

$$\begin{aligned} \phi_{\text{even}}(x, y) &= e^{-r^2/\sigma^2} \cos(\omega x') \\ \phi_{\text{odd}}(x, y) &= e^{-r^2/\sigma^2} \sin(\omega x') \end{aligned} \tag{16}$$

where $r^2 = x'^2 + y'^2$ and σ and ω are parameters. The exponential term is a Gaussian envelope that localises the feature extractor. Coordinates x' and y' are transforms of x and y, shifted and rotated by angle θ:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta & x_0 \\ -\sin\theta & \cos\theta & y_0 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{17}$$

Note that y' is absent from Equation 16 so that the function represents a two-dimensional sinusoid, offset by (x_0, y_0), localised in space by the Gaussian, and oriented at angle (θ).

Thirty-two such functions were used; the set of the five parameter values was chosen empirically on the grounds of efficacy and is as follows. Assume the N $\times$ N image has its origin at its center (0,0). Use identical Gabor functions in each of the four quadrants centered at positions $(\pm N/4, \pm N/4)$. Within each quadrant, use four orientations at angles 0, $\pi/4$, $\pi/2$, and $3\pi/4$ radians. Use one spatial frequency, $\omega = 2\pi\sqrt{2}/N$, and one Gaussian envelope radius, $\sigma = N\sqrt{2}/8$. Use both even (cosine) and odd (sine) functions.

Approximate the image by a linear combination of the 32 Gabor functions; obtain the coefficients by a least-squares fit. These 32 coefficients are the extracted features in the FL3 data set; they are the inputs to the multilayer perceptron networks in the next section.

4. Results

The initial weights were chosen from a uniform random distribution in the range $(-0.5, +0.5)$. Since different minima are found with different starting weights, a statistical sampling of runs is necessary, each run with a different seed to start the random number generator. Each run finds some local minimum, but the particular one found is a sensitive function of the initial choice of weights. In fact, two different brands of computers will ordinarily not find the same local minimum, even given the same initial choice of weights; extremely minor differences in doing arithmetic are sufficient, over many iterations, to cause different local minima to be found.

All the runs use the same classification rules to divide the results into three categories: Right, Unknown, and Wrong. For each pattern, the highest of the 10 output values is found. If the activation level is below a set activation threshhold, the result is defined to be Unknown. If the activation level at least as large as the threshhold, the result is accepted, and the answer is either Right or Wrong. Except for Figure 3, all results reported here use a zero activation threshhold, eliminating Unknowns.

For both BP and SCG, by far the most time-consuming part of the calculation is done by the **forward** subroutine, which feeds the inputs forward to the outputs according to Equation 1, then feeds outputs backwards according to Equation 7 to produce the gradient of the error function (step 4 of the algorithm in Section 1.1). BP does this once per iteration; SCG usually does this twice per iteration, but occasionally more than once. For comparing BP and SCG, we therefore compare the number of calls to **forward** rather than the number of iterations.

Comparisons of algorithms depend strongly on the stoppping criteria described in Section 2. The runs presented here use one of two sets of stopping criteria. The *early* set uses criterion 5, stopping when the number of correctly classified training patterns increases too slowly. Criteria 2, 3, and 4 are disabled by setting C_2, C_3, and C_4 to zero. (Criterion 4 is not quite disabled by $C_4 = 0$; if a local minimum is found, the error cannot be decreased further.) Good values for SCG are $C_5 = 1$ and $K = 10$; $C_1 = 1000$ is safe, since at most a few hundred iterations are necessary. For BP it is necessary to use a larger K, such as $K = 100$, and to allow a larger number of iterations. The *early* set strikes a reasonable balance between the level of convergence obtained and the computer time used.

The *late* set emphasizes the level of convergence at the expense of computer time; it uses criterion 1, stopping when the maximum number of iterations has been used. Criteria 2, 3, and 4 are disabled as in the *early* set, with zero values for C_2, C_3, and C_4. Criterion 5 is disabled by $C_5 = -100$. Reasonable values for SCG are $K = 10$ and $C_1 = 1000$. BP needs $K = 100$ and $C_1 = 10000$ or more.

In both sets, instead of $C_3 = 0$, a value like $C_3 = 10^{-12}$ may be used for quicker stopping in case of accidentally finding a local minimum quickly.

4.1. Training on the training set

Training on the 2000-pattern "training set," followed by testing on the 1434-pattern "testing set," gives a realistic picture of what can be expected. Note that, for a practical application, only the best set of weights found is important, since any inferior sets will be discarded.

Table 1 illustrates the effect of changing the number of hidden nodes in the network. Eight hidden nodes are clearly insufficient, but after 24 hidden nodes the improvement is small. In general, as the number of hidden neurons increases, training and testing improve somewhat, training is more likely to find an inferior local minimum, and training takes more calls to **forward**.

Choosing the Levenberg-Marquardt parameter, μ, must be done experimentally; $\mu = 10^{-3}$ is reasonable; this value does not minimize the $\mathcal{E}$ error, but it gives better classification results on both the training and the testing sets.

Figures 1 (early stopping criteria) and 2 (late stopping criteria, $C_1 = 1000$) illustrate the variability of results. Each figure shows results from computer runs starting from 50 different random seeds for $\mu = 0$ and for $\mu = 10^{-3}$. The activation threshhold is zero, so that there are no Unknowns. Note that $\mu = 10^{-3}$ is slightly better than $\mu = 0$ and that late stopping gives slightly better results than early stopping, although at a significant penalty in computer time: the median number of calls to **forward** is 303 for early stopping and 2002 for late stopping. Also note that there is little correlation between the performances on the training set and on the testing set.

Typical results for percent correct on the testing set as a function of the number of Unknowns is shown in Figure 3. For each run, the activation threshhold was successively changed to generate the desired percent of Unknowns; then the remainder of the test pattern results were classified as Right or Wrong. For each value of $N^{(1)}$, ten random starts were done and the run with the best number correct at zero activation threshhold was chosen. Note that little improvement is seen for $N^{(1)} > 16$.

4.2. Training on the full set

To demonstrate that it is possible to have a set of weights that does equally well on both sets, some runs trained on all 3434 patterns. As an example, with 24 hidden nodes, training produced a set of weights which gave 98.9% right on the full set and 99.1% on the testing set.

4.3. Comparison with backpropagation

In order to compare CG and BP, reasonable values of η and α must be found. Because of the $1/N^{(P)}$ in the definition of $\mathcal{E}(\mathbf{w};\mu)$, our η is larger by a factor of $N^{(P)}$ than the usual one, and is to first order independent of $N^{(P)}$. Table 2 gives the training error after 500 iterations of BP using the first 500 patterns of the training set, all with the same random number seed. Note that 500 iterations (501 calls to `forward`) are insufficient to train the 32-16-10 network.

From this survey and similar ones, reasonable values are $\eta = 10$ and $\alpha = 0.25$. (No claim is made that these are ideal values.)

Figures 4 and 5 compare single runs of BP and SCG for a 32-24-10 network, with late stopping and $\mu = 10^{-3}$, with $K = 10$ for SCG and $K = 100$ for BP. Note that any stopping criterion based on slowness of decrease of the error or of the number wrongly classified is likely to stop well before the lowest values are obtained.

5. Conclusions

SCG is faster than BP. An exact comparison is hard to obtain, but SCG appears to be about 10 times faster than BP.

FL3 is a reasonable data set to use, but is not a large enough set for training a practical network for classifying handprinted digits. If it were large enough, the recognition rate on the testing set would not be appreciably lower than the rate on the training set.

References

[1] D. L. Alkon, K. T. Blackwell, G. S. Barbour, A. K. Rigler, , and T. P. Vogl. Pattern-recognition by an artificial network derived fron biological neuronal systems. *Biological Cybernetics*, 62:363–376, 1990.

[2] G. A. Carpenter and S. Grossberg. Art 2: self-organization of stable category recognition codes for analog input patterns. *Applied Optics*, 26:4919–4930, 1987.

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, et al., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations*, chapter 8, pages 318–362. MIT Press, Cambridge, 1986.

[4] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Englewood Cliffs, NJ, 1983.

[5] E. Stiefel. Uber einige methoden der relaxationsrechnung. *Z. Angew. Math. Physik*, 3:1, 1952.

[6] H. Akaike. On a successive transformation of probability distribution and its application to the analysis of the optimum gradient method. *Annals of the Institute of Statistics and Mathematics, Tokyo*, 11:1–16, 1959.

[7] E. Polak and G. Ribière. Note sur la convergence de methodes de directions conjugées. *Rev. Française Information Recherche Operationnelle*, 16:35–43, 1960.

[8] R. Fletcher and C. M. Reeves. Function minimization by conjugate gradients. *Computer Journal*, 7:149–154, 1964.

[9] E. M. Johansson, F. U. Dowla, and D. M. Goodman. Backpropagation learning for multi-layer feed-forward neural networks using the conjugate gradient method. *IEEE Trans. on Neural Networks*. To be published.

[10] M. F. Møller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*. To be published.

[11] C. L. Lawson and R. J. Hanson. *Solving Least Squares Problems*. Prentice-Hall, Englewood Cliffs, NJ, 1974.

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

[13] D. F. Shanno and K. H. Phua. Remark on algorithm 500: Minimization of unconstrained multivariate functions. *ACM Trans. on Mathematical Software*, 6:618–622, 1980.

[14] P. M. Flanders, R. L. Hellier, H. D. Jenkins, C. J. Pavelin, and S. Van Den Berghe. Efficient high-level programming on the amt dap. *IEEE Proceedings: Special Issue on Massively Parallel Computers*, 79(4):524–536, April 1991.

[15] C. L. Wilson and M. D. Garris. Handprinted character database. *National Institute of Standards and Technology*, Special Database 1, **HWDB**, April 18, 1990.

[16] R. A. Wilkinson. Segmenting of text images with massively parallel machines. In *Visual Communication and Image Processing*, Boston, MA, October 1991. SPIE.

[17] D. Gabor. Theory of communication. *Journal of the Institute Electrical Engineers*, 3(93):429–457, 1946.

[18] J. G. Daugman. Representational issues and local filter models of 2d spatial visual encoding. In D. Rose and V. G. Dobson, editors, *Models of the Visual Cortex*, pages 96–107. J. Wiley and Sons, 1985.

[19] M. D. Garris, R. A. Wilkinson, and C. L. Wilson. Analysis of a biolgically motivated neural network for character recognition. In *Proceedings: Analysis of Neural Network Applications*, pages 160–175. ACM Press, May 1991.

[20] M. D. Garris, R. A. Wilkinson, and C .L. Wilson. Methods for enhancing neural network handwritten character recognition. In *International Joint Conference on Neural Networks*, volume 1, pages 695–700. IEEE and International Neural Network Society, July 1991.

Table 1: Results of training 32-$N^{(1)}$-10 feed-forward networks with SCG, $\mu = 10^{-3}$, and early stopping. Tabular values are the number of calls to **`forward`**, the percent correct on the training set, and the percent correct on the testing set, for the best run, selected from 10 runs with random starts.

$N^{(1)}$	Calls	Training	Testing
8	321	94.9%	93.0%
16	321	98.1%	95.1%
24	322	98.8%	95.2%
32	422	99.3%	95.7%
40	401	99.4%	95.3%
48	507	99.2%	95.2%

Table 2: Results of training a 32-16-10 feed-forward network with 500 iterations of backpropagation on the first 500 patterns of the training set with seed 12345 and various values for η and α. The tabular value is the error.

η	α				
	0.10	0.25	0.50	0.75	0.90
1	0.269	0.263	0.247	0.203	0.260
2	0.240	0.229	0.196	0.162	0.273
5	0.166	0.153	0.131	0.238	0.285
10	0.124	0.116	0.196	0.285	0.316
20	0.197	0.222	0.285	0.316	0.316

Figure 1: Scatter plot of testing results vs. training results for 32-24-10 networks, early stopping. Open circles: $\mu = 0$; filled circles: $\mu = 10^{-3}$.

Figure 2: Scatter plot of testing results vs. training results for 32-24-10 networks, late stopping. Open circles: $\mu = 0$; filled circles: $\mu = 10^{-3}$.

Figure 3: Percent correct as a function of the percent Unknown. 32-$N^{(1)}$-10 networks, early stopping, $\mu = 10^{-3}$. filled circles: $N^{(1)} = 8$; filled triangles: $N^{(1)} = 16$; open circles: $N^{(1)} = 24$; open triangles: $N^{(1)} = 32$.

Figure 4: Error vs. number of function calls for a 32-24-10 network, late stopping, $\mu = 10^{-3}$. filled circles: SCG; open circles: BP.

Figure 5: Percent wrong vs. number of function calls for a 32-24-10 network, late stopping, $\mu = 10^{-3}$. filled circles: SCG; open circles: BP.