EFFICIENT TRAINING OF THE BACK PROPAGATION NETWORK BY SOLVING A SYSTEM OF STIFF ORDINARY DIFFERENTIAL EQUATIONS

A. J. Owens and D. L Filkin
Central Research and Development Department
P. O. Box 80320
E. I. du Pont de Nemours and Company (Inc.)
Wilmington, DE 19880-0320

Abstract. The training of back propagation networks involves adjusting the weights between the computing nodes in the artificial neural network to minimize the errors between the network's predictions and the known outputs in the training set. This least-squares minimization problem is conventionally solved by an iterative fixed-step technique, using gradient descent, which occasionally exhibits instabilities and converges slowly. We show that the training of the back propagation network can be expressed as a problem of solving coupled ordinary differential equations for the weights as a (continuous) function of time. These differential equations are usually mathematically stiff. The use of a stiff differential equation solver ensures quick convergence to the nearest least-squares minimum. Training proceeds at a rapidly accelerating rate as the accuracy of the predictions increases, in contrast with gradient descent and conjugate gradient methods. The number of presentations required for accurate training is reduced by up to several orders of magnitude over the conventional method.

Introduction and Basic Concepts

Artificial neural networks are mathematical models, usually formulated and solved on conventional digital computers, which were originally designed to mimic some functions of the nervous systems of higher animals. Although inspired by physiological and neurophysiological research, the various neural network architectures that have been proposed can be viewed as computing paradigms in their own right. Here we discuss one class of artificial neural network -- the back propagation network [BPN] -- as a computational model. We develop a new BPN training algorithm, based on the formulation and solution of the problem as a set of stiff ordinary differential equations.

Most artificial neural network paradigms are built around the concept of adjustable weights connecting simple processing elements often referred to as nodes. As an example, Figure 1 shows the relationship between a set of five input nodes, P, and a single output node, Q.

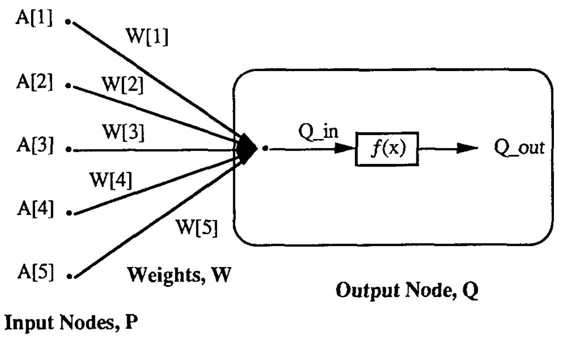

Figure 1. Relationship Between Input Nodes P and An Output Node Q In A Neural Network

The input nodes in this figure are labeled P[i], where i = 1,2,...5, and the single output node is denoted by the symbol Q. The node P[i] has a signal strength or activation denoted by A[i]; usually the activations are limited to the range $0 \leq A[i] \leq 1$.

The single node Q, enclosed in the box, receives as its input the inner product given by the sum of the activations A[i] of the nodes P[i] multiplied by the corresponding connection weights (W[i]) which relate node P[i] to Q:

$$\text{Input to Node Q} = Q\_in = \sum_{i=1}^{5} W[i] * A[i] \tag{1}$$

The output signal from node Q is obtained by passing the inner product Q_in through a nonlinear (or semilinear) scalar function f:

$$\text{Output from Node Q} = Q\_out = f(Q\_in) \tag{2}$$

The semilinear function $f(x)$, sometimes called the "squashing function," is an increasing function of x that has a limited range of output values. A commonly used squashing function is the sigmoid

$$f(x) = 1/(1+\exp\{-(x+\phi)\}) \tag{3}$$

where ϕ is called the threshold of the node. The squashing function has limits $0 \leq f(x) \leq 1$, which guarantees that the output of node Q is limited to the range $0 \leq Q\_out \leq 1$, regardless of the value of Q_in. The threshold is equivalent to the weight from a node in the previous layer with an activation always equal to 1. As is conventional, in the discussion below the thresholds will be included implicitly as such weights.

A single node thus performs an inner product of the connection weights and activations from its input connections and passes the result through a nonlinear squashing function. The output is then "fanned out" to the nodes in the subsequent layer. In an artificial neural network, several layers of these computational elements are connected. Each class of artificial neural network differs in the topology of the connections, the choice of the squashing function $f(x)$, and the "training rule" by which the weights W are selected.

Learning by Gradient Descent

The automatic learning rules for artificial neural networks give them most of their unique capabilities. Many of the networks "learn" by using the concept of gradient descent of the prediction error in a search space spanning the adjustable weights. In the most commonly applied version of the back propagation network [BPN], as discussed for example by Rumelhart et al. [3], the network is presented sequentially with a number of "training patterns" relating the activations of the input nodes to the corresponding values on the output nodes. All the connection weights are then varied to minimize the (squared) error in the network's predicted outputs.

To understand the spirit of this technique, consider the simple two-layer example shown in Figure 1, and suppose that there is only one training pattern. The training pattern associates one specified set of input signals A(1), ... A(5) with a single output, Q_out. At any given stage in the training of the network weights W[i], the error E to be minimized is the squared difference between the desired output Y and the network value Q_out:

$$E = (Y - Q\_out)^2 \tag{4}$$

To change the weights W[i] via the method of steepest descent, find the gradient of E with respect to W[i], and then change W[i] to move E toward smaller values. Thus to train the network using gradient descent, change the weights according to the formula

$$\Delta W[i] = - \partial E/\partial W[i] \qquad (5a)$$

Applying the chain rule of calculus,

$$\Delta W[i] = - (dE/dQ\_out)\,(dQ\_out/dQ\_in)\,(\partial Q\_in/\partial W[i]) \qquad (5b)$$

With the linear inner product rule for weights (Equation 1) plus the sigmoid squashing function (Equation 3), this expression gives

$$\Delta W[i] = 2\,[(Y - Q\_out)]\,[Q\_out\,(1 - Q\_out)]\,A[i] \qquad (6)$$

For several output nodes, since there is a unique connecting weight between each input and output node, the same reasoning can be applied to find the change in the weight W[i,j] connecting input node P[i] with output node Q[j]. Generalization of equation (6) to the case in which several training patterns are used results in a sum over all the training patterns. This method for updating the weights is called as the "delta rule" in [3]. It is due to Widrow [4], who calls the method least mean squares or "LMS".

The Conventional BPN Training Algorithm

The topology of the conventional back propagation network, illustrated in Figure 2, comprises input nodes, one (usually) or more layers of interior or "hidden" nodes, and output nodes. Typically the network is fully connected, so that nonzero weights relate all the input-to-hidden nodes and all the hidden-to-output nodes, and the sigmoid squashing function of equation (3) is used.

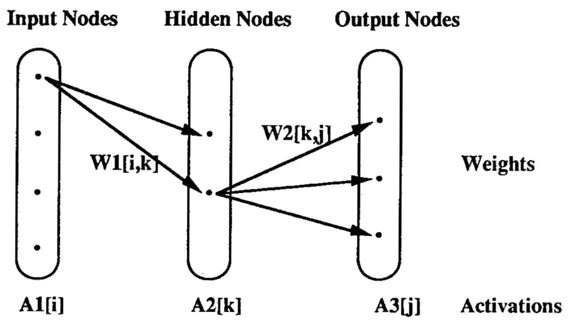

Figure 2. Topology of a Three-Layer Back Propagation Network (With One Hidden Layer)

The delta training rule discussed in the last section is applicable to the case of a simple perceptron-like input-output system, with no hidden nodes. (It is also applicable for adjusting the weights between the last hidden layer and the output layer in the BPN.) The BPN models have one or more hidden layers, and the weights to these layers must also be adjusted. This weight change is computed by successive application of the chain rule, beginning at the output layer and "propagating backward" the prediction errors to subsequent layers through the input layer. (Hence the network's name.)

The derivation of the weight changes required for gradient descent is given in some detail by Rumelhart et al. [3], where the result is called the "generalized delta rule". We will denote these weight changes as ΔW[gd]. (Here [gd] signifies gradient descent.) They are the generalization of equation (6) to neural networks with multiple outputs and training patterns, as well as with one or more hidden layers.

The conventional method for training the BPN can now be described. The weights are initialized to small random values. All of the training patterns are presented to the network, and the changes in the weights required for gradient descent are computed to give ΔW[gd]. The weights are then changed, updating all of them together, by the iterative rule [3]

$$W^{n+1} = W^{n} + \eta\,\Delta W[gd] + \alpha\,(W^{n} - W^{n-1}) \qquad (7)$$

Here W^n is the weight at iteration number n, and W^{n-1} is the weight at the previous iteration. For clarity, in equation (7) the label identifying the layer number and the subscripts for the node connections are suppressed; η is called the learning rate, and the last term on the RHS is the momentum term. Both η and α are problem-dependent adjustable parameters, typically with magnitudes smaller than unity. In the discussion below, training following equation (7) will be denoted the traditional method.

In this method, training occurs in discrete steps or iterations, each requiring one presentation of all the training patterns to the network. To ensure convergence of the iterative algorithm, the learning rate η must be chosen small enough. If it is too large, oscillations and instabilities occur. Most applications require the training rate η to be less than one. The momentum term α, which is used by some researchers but omitted by others, is supposed to increase the convergence rate and help avoid the problem of being trapped in local minima of the least-squares surface.

Pictorially, imagine being placed somewhere on a complex N-dimensional surface, where N is the number of adjustable weights and thresholds, with the goal of finding the lowest point in the nearest valley. The surface represents the sum of squares of prediction errors as a function of these weights. To find the minimum using the generalized delta rule, first face downhill, and then take a step of fixed length in that direction. (If momentum is present, your step will also include a fraction of the step you just took previously.) Repeat this process (face downhill, take a fixed-size step) until you reach a minimum.

Although the training algorithm in equation (7) gives a change in the weights that corresponds approximately to the direction of steepest descent, it does not always converge. (E.g., the step may be too large, and it takes you up the other side of a valley rather than downhill.) Perhaps more problematic, the training is initially fairly rapid but eventually becomes extremely slow, especially as the network becomes more accurate in predicting the outputs. Training to give highly accurate analog predictions can take thousands or even millions of presentations of the training set.

New "Stiff" Training Algorithm

While watching the training of the BPN using the conventional algorithm, we were impressed with the similarities between the temporal history of the weight changes and those of ordinary differential equation systems that are stiff. Stiff systems are characterized by time constants that vary greatly in magnitude. In the BPN, this corresponds to the fact that some weights reach a near-steady-state value very rapidly while others change only very slowly. Sometimes a weight in the BPN will remain unchanged for a large number of iterations and then suddenly change to another quasi-steady value; this behavior is a common occurrence for variables in stiff differential systems. Finally, solution schemes for neural networks involve heuristics for adjusting η and α to try to obtain rapid but stable convergence, just as mathematicians struggled for algorithms to solve stiff systems before stiff differential equation solvers first became available [1].

The fundamental training concept of the BPN is to change the weights in such a way as to decrease the squared error of the output predictions. Instead of the usual BPN algorithm discussed in the previous section, in which one thinks of the training as taking place in discrete steps labeled n, imagine the training to take place as a (continuous) time evolution of the weights W.

To illustrate the technique, we return to the discussion of the training of the simple system shown in Figure 1. In place of the discrete equation (5), we consider the weights W[i] to be functions of time. To change the weights so that the squared prediction error E decreases monotonically with time, the weights are required to move in the direction of gradient descent. Thus we replace the discrete-step equations (5a) with the ordinary differential equations (ODE's)

$$dW[i]/dt = - \partial E/\partial W[i] \qquad (8)$$

At any given time t, the evaluation of the gradient $\partial E/\partial W[i]$ follows exactly the same steps as outlined above. The generalized delta rule applies for the three-layer BPN, exactly as discussed above, giving the schematic equation

$$dW[i]/dt = \Delta W[i; gd] \qquad (9)$$

where $\Delta W[i; gd]$ represents the weight changes given in equation (6). There is one ODE for each adjustable weight W[i], and the right-hand sides are nonlinear functions of all the weights.

Comparing equations (7) and (9) shows that each unit of time t in our differential equations corresponds to one presentation of the training set in the conventional algorithm, with a training rate $\eta = 1$ and no momentum.

Although the dynamical properties of neural networks have previously been discussed in terms of differential equations like (8) (e.g., [2]), the dynamical behavior near steady-state attractors has been of interest. Our novel points here are that we discuss these differential equations far from equilibrium, and that we present an efficient BPN training algorithm based on the direct solution of these equations using a stiff ODE solver.

In generalizing from the simple portion of a neural network in Figure 1 to a complete back propagation network like that given in Figure 2, exactly the same steps are applied as in the discussion above. Thus, for training a BPN with an arbitrary number of hidden layers, we propose using the differential equation in the form of equation (9) rather than the discrete-step form of equation (7).

In the absence of momentum (i.e., $\alpha = 0$), the conventional discrete-step iterative scheme of equation (7) is *equivalent* to a fixed-time-step forward-Euler method (with time step η) for integrating the underlying differential equations (9). Since the differential equations of the BPN are usually stiff, as we will discuss below, a fixed-step forward-Euler integrator is both an inefficient and a potentially unstable method for solving the training equations.

The *stiff stability* of numerical methods for solving ordinary differential equations is discussed in detail by Gear [1]. The Hessian matrix

$$H[i,j] = -\partial^2 E / \partial W[i]\, \partial W[j]$$

of the system plays a critical role in the analysis of stiff stability. [In differential equation literature, the Hessian matrix is called the Jacobian matrix.] All *explicit* numerical solution schemes have a limiting step size for stiff stability which is proportional to $1/\Lambda$, where Λ is the largest eigenvalue of the Hessian matrix. Such schemes include the forward-Euler method and higher-order related methods, of which the traditional BPN training algorithm (both with or without momentum) is a subset. The larger the range of eigenvalues of the Hessian matrix, the greater is the stiffness of the underlying ODE system. For a related class of problems, the degree of stiffness usually grows with the size of the problem.

In terms of the traditional BPN training, with a large number of adjustable weights, the stiffness of the system limits the training rate η to small values, and training progresses slowly. Stiff ODE solvers, in contrast, are constructed so that there is no limiting step size for stability.

For the three-layer BPN as illustrated in Figure 2, we solve the differential equations

$$dW1[i,k]/dt = \Delta W1[i,k; gd] \qquad (10a)$$

$$dW2[k,j]/dt = \Delta W2[k,j; gd] \qquad (10b)$$

where $\Delta W1$ and $\Delta W2$ are exactly the same gradient-descent "generalized deltas" used on the right-hand side of equation (7) in the traditional approach [3]. These are a set of coupled nonlinear ordinary differential equations for the continuous variables W1[i,k] and W2[k,j], with one equation for each adjustable weight or threshold. The initial conditions are small random weights. This system of equations is integrated numerically, using the stiff integration package DGEAR (recently renamed IVPAG) from the IMSL library (IMSL, Inc., Houston, Texas) to solve the ODE's. As time increases, the weights approach steady-state values corresponding to those giving a (local) minimum least-squares prediction error for the training set.

Because the differential equations (10) are formulated to decrease the squared prediction error in the direction of steepest descent (see equation 8), the prediction error always decreases. Modern stiff differential equation solvers [1] are "A-stable", so that the stability of the solution is not limited by the computational step size taken. There are thus no convergence or oscillation problems. As in other stiff systems, the step size initially taken by the integrator is small (less than one), but it increases steadily as the rapidly decaying eigenfunctions die away and the solution approaches the equilibrium state of minimum prediction errors. When some weights change rapidly midway through the training set, as often occurs in BPN training (e.g., in the parity-recognition problem), the integrator automatically reduces its step size and re-evaluates the Hessian matrix if necessary.

The stiff equation solver computes and makes use of the Hessian matrix H[i,j], which includes information about the local curvature of the error surface. This requires the storage of the order of N*N real numbers, which restricts the size of problems to less than about 1000 adjustable weights on real-memory computers. It also involves a startup cost, since (N+1) initial gradient evaluations must be made to develop H[i,j]. The use of this curvature information, however, allows the method to take large computational steps, even when the error surface does not have a simple shape.

Pictorially, imagine once more being placed on an N-dimensional complex surface with the goal of finding the bottom of the nearest valley. Begin walking downhill, always observing the local terrain. If a long step still takes you downhill, take it. If the terrain is convoluted, take smaller steps, each one in the downhill direction, which is continuously changing as you walk.

The BPN Training Equations *Are* Stiff

As discussed above, a system of ODE's is stiff if the ratio of the eigenvalues of the Hessian matrix H[i,j] is large, and the limiting step size for explicit solution methods (e.g., the traditional BPN training method) is proportional to $1/\Lambda$, where Λ is the largest eigenvalue of the matrix H. To evaluate the stiffness of a BPN, the Hessian can be computed at several points during the training, and the eigenvalues computed with a standard mathematical subroutine (e.g., ELVSF from IMSL).

A pictorial representation of stiffness in the training of the BPN is given in Figure 3, produced by R. J. Hubner (Du Pont Engineering Physics Laboratory). In this figure, time (i.e. number of training presentations) is represented in the horizontal direction. The graph shows the weights connecting all the nodes in the BPN for the classification problem discussed below. A traditional BPN training algorithm (equation 7) was used. The vertical width of the lines represent the amount the weights changed over the presentation of all the patterns in the training set. A fully converged solution would result when all the lines in the figure are horizontal and thin.

Figure 3 shows that many of the weights reach their nearly converged values quickly. However, several of the weights [especially those with values outside the normal range] change very slowly with time. Some of the weights appear to be at steady values, but then they change rapidly again at some later point in the training. The maximum training rate for stability was about 1/4, but some of the weights are still changing substantially after the 600 training iterations shown. These features show that the weights are evolving according to differential equations that are mathematically stiff.

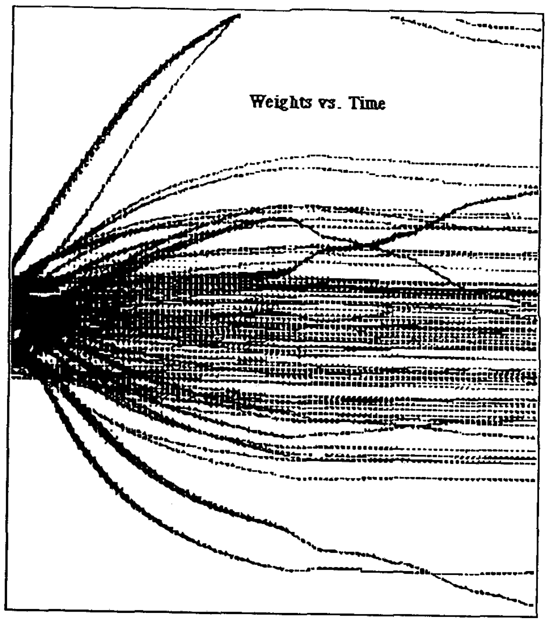

Figure 3. Connection Weights as a Function of Time for a Back Propagation Network Illustrating Stiff Behavior.

It is well known in the solution of ordinary differential equations that the use of a nonstiff integrator (e.g., a forward-Euler or Runge-Kutta method) on a stiff system can be extremely inefficient, even if the methods include an automatic (adaptive) step-size adjustment. As discussed above, the conventional BPN training rule (equation 7) is essentially a fixed-step forward-Euler integrator for the differential equations (8). It is the *simplest* known integrator. When applied to stiff systems, it is also probably the *least efficient*.

Whether the training equations are in fact *stiff* depends, of course, on the topology of the neural network and on the training data. In addition to computing the eigenvalues of the Hessian matrix, the degree of stiffness of a system is easy to evaluate in a practical way, at least *ex post facto*. There is an option in most commercially available stiff ODE solution subroutines (e.g., from IMSL and Numerical Algorithms Group) to run in both stiff and nonstiff modes. The stiffer the system, the more rapidly the stiff integrator works compared with the nonstiff one. For very stiff systems, speedups of thousands or more over the best nonstiff methods are possible [1].

We have investigated a large number and a wide variety of BPN training tasks, including classical problems (e.g., exclusive-or, parity) and real-world applications (quasi-statistical data analysis). Some involve binary inputs and/or outputs, while many others have analog signals. In all these sample problems, the new algorithm trained to a specified accuracy more quickly than the conventional one, by factors ranging from about 2 (small problems, low-accuracy fitting) to more than 1000 (large problems, very accurate predictions required).

We usually train the BPN using our "stiff" method with intermediate printout "time" steps (corresponding to number of training-set presentations in the conventional algorithm) increasing by a factor of two: 100, 200, 400, 800, ... Monotonic convergence is achieved in each case: the total prediction error decreases continuously to the nearest local minimum value.

The Exclusive-Or (XOR) Problem

The exclusive-or problem (XOR) is a classic BPN benchmark. Two binary (i.e., *false* or *true*) inputs are presented to the network, and the output is trained to reproduce the logical *exclusive-or* truth table. For two hidden nodes, the performance of the stiff ODE training method is compared with the nonstiff and traditional methods in Figure 4. For this typical set of random initial conditions, convergence to a tolerance of 0.05 (i.e., 5%) took 250 and 438 presentations of the 4-member training set for the stiff and nonstiff integrators, respectively, indicating a moderate degree of stiffness. The traditional method required more than 3000 such training presentations, about 15 times as many as the stiff ODE training method. Similar graphs result from other starting conditions.

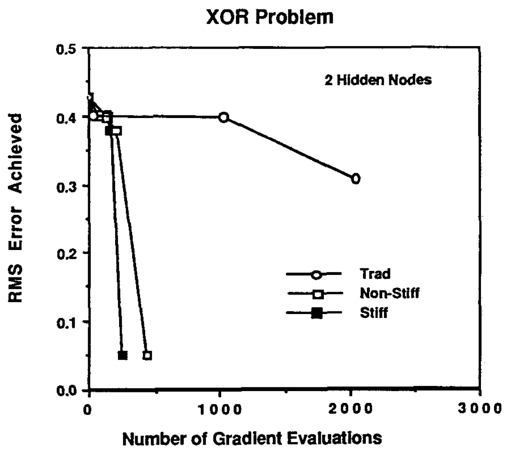

Figure 4. Number of Gradient Evaluations Required to Reach a Given Training Accuracy (RMS Prediction Error) for Several BPN Training Algorithms. For the [Trad] Traditional Training Rule, $\eta = 0.25$ and $\alpha = 0$.

A Classification Example

For most BPN applications, the training data set determines the number of input and output nodes. We normally preprocess the data by scaling the inputs and outputs individually and linearly in both the training and the prediction data sets to lie in the range 0.1 to 0.9. In this example, data from 11 physical measurements of a manufactured part are to be related to the visual inspector's classification into the two quality categories of reject or accept [coded as 0.1 and 0.9]. The training set consists of 42 samples, of which 36 samples were accepted and 6 rejected. This case has 11 input nodes, one for each of the physical measurements, and one output node, whose value designates acceptance or rejection. [The scaled training data are available from the authors on request.]

If a BPN has n_i input nodes, n_h hidden nodes, and n_o output nodes, then the total number of adjustable weights and thresholds is

$$n_t = (n_i + 1)*n_o + (n_h + 1)*n_o \qquad (11)$$

If n_t is small compared with the number of training sample outputs, the network is "interpolative" in nature, and subtle variations in the input data cannot be mapped to output changes. If n_t is large, the network approaches the "pattern recognition" mode in which it is memorizing each input pattern. If a few more internal nodes are specified than needed for accurate classification, the network often tends to a solution in which several nodes are either "on" of "off" for all the training patterns. These redundant nodes can be removed from the system. If, on the other hand, too few hidden nodes are specified, the network converges slowly, and it misclassifies some of the members of the training set.

With our new convergence algorithm, there are only *two* adjustable parameters in the training of the BPN. First, the accuracy of training (i.e., the residual prediction error of the outputs in the training set) must be specified. Two convergence criteria are used, and the acceptance of either one causes the training to stop: (a) if the root-mean-square (RMS) error in predicting the outputs is less than the user-specified tolerance TOLER (in scaled units), and (b) if none of the individual output errors exceeds 3*TOLER. For the classification example, we chose a tolerance of 0.01 (i.e., 1%) to assure that all classification decisions in the training set were correctly modeled.

The second user-adjustable parameter is the number of internal or hidden nodes used in the neural network, which determines the number n_t of adjustable weights and thresholds as in equation (11).

For this classification example, we initially specified four hidden nodes. Two of these were redundant: after training, their activations were within 0.001 of 1 across all the 42 training patterns. The example output in Printout 1 gives the results with the number of hidden nodes reduced to two. With 42 training patterns and 27 adjustable weights and thresholds, there were 15 remaining degrees of freedom. The neural network, with only two hidden nodes, was able to correctly classify all 42 accept/reject decisions.

Convergence to a tolerance of 0.01 required less than seven seconds of execution time on a CRAY X-MP/24 computer, using 25,600 time steps. If the traditional training algorithm converged accurately with $\eta = 1$ and $\alpha = 0$, it would require 25,600 presentations of the training patterns to reach this solution. [In fact, for this problem, learning rates above approximately 0.25 were unstable.] The stiff training run illustrated in the printout, however, required only 500 training presentations.

Printout 1. Output File from training the BPN for the CLASSIFICATION EXAMPLE on the Du Pont CRAY X-MP/24

```
  There are   42 training patterns ...

    Number of adjustable weights = 27

Number of    Fraction of       RMS error         Calculation
Iteration    Wrong (>3*TOLER)  of Predictions      Step Size
Time Steps   Predictions

    100      0.64               0.2195               4.66
    200      0.74               0.1837               4.61
    400      0.71               0.0625              13.00
    800      0.57               0.0445              46.66
   1600      0.29               0.0317              88.73
   3200      0.10               0.0227             204.07
   6400      0.02               0.0185             398.08
  12800      0.02               0.0167             866.84
  25600      0.02               0.0097             207.77

  -- Program exited after 25600 iterations --
```

The monotonic convergence of the RMS prediction error is illustrated in the sample printout, along with the rapid buildup of the computational step size being used by the algorithm. Toward the end of the training (see the underlined numbers), the new algorithm is learning at a rate hundreds of times faster than the traditional iteration scheme.

Figure 5 shows the performance of the stiff training algorithm for this problem, compared with a nonstiff Adams method (IMSL's nonstiff option in DGEAR) for solving equation (9) and with the traditional fixed-step method of equation (7). The calculation of the gradient of the weights via the generalized delta rule dominates the computation, with one gradient evaluation per presentation of all the training patterns, so the horizontal axis is proportional to the computing time required. For training past a few hundred presentations, the stiff method rapidly becomes more efficient than the traditional one.

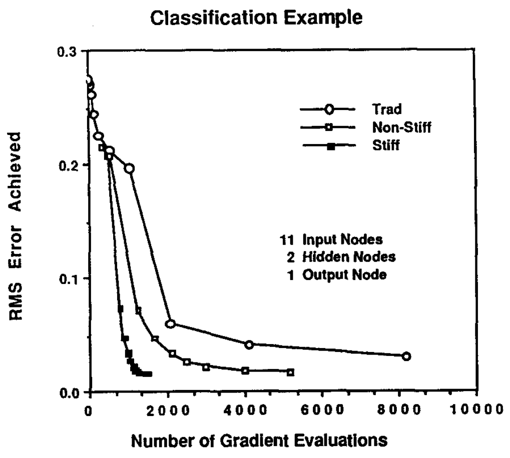

Figure 5. Number of Gradient Evaluations Required to Reach a Given Training Accuracy (RMS Prediction Error) for Several BPN Training Algorithms. For the [Trad] Traditional Training Rule, $\eta = 0.25$ and $\alpha = 0$.

A moderate degree of stiffness in the system is shown by the fact that the nonstiff integrator, even though having an efficient algorithm with self-adjusting step size, takes longer than the stiff integrator to reach a specified level of error in predicting the outputs for the training set:

RMS Error Achieved	Ratio of Number of Gradient Evaluations Required Nonstiff/Stiff	Trad/Stiff
10%	1	1
5%	2	4
2%	5	>20

The results in Figure 5 and this table are based on training from the same initial conditions of small random weights, which yielded an initial RMS prediction error of 27%. Qualitatively similar results are obtained from other starting weights (as in Printout 1). The values of the training rate used in the traditional method ($\eta = 0.25$) was the largest giving stable convergence.

Discussion

The back propagation network is well suited for solving a wide variety of formulation, classification, process control, and pattern-recognition problems. The BPN is a useful complement to traditional statistical methods, particularly when the underlying interactions are complex and nonlinear. Compared with multiple linear regression, for example, the neural network can accommodate more complex nonlinear relationships between inputs and outputs. The underlying principle of minimizing the squared prediction error is the same, but the BPN is more general. The BPN is also model-free, in contrast to both linear and nonlinear least-squares fitting procedures, since no assumption is made about the inherent functional form relating inputs to outputs. The accuracy of the BPN representation is limited only by the quality of the data, not by the adequacy of the model.

For these reasons, given a sufficiently complete data base, the creation of a BPN solution for a wide range of practical problems can usually be completed more quickly and easily than traditional statistical models. With our new "stiff" training algorithm, the utility of the BPN for these applications is enhanced by adding ensured parameter-free convergence and much more rapid training. The use of the BPN then moves from a research method toward becoming a widely applicable engineering tool.

References

[1] Gear, C. W., 1971, Numerical Initial Value Problems in Ordinary Differential Equations, Prentice-Hall, Englewood Cliffs, NJ.

[2] Pineda, F. J., 1988, Generalization of backpropagation to recurrent and higher order neural networks, in Neural Information Processing Systems, D. A. Anderson, ed., Denver, Co, 1987, American Institute of Physics, New York, pp. 602-611.

[3] Rumelhart, D. E., G. E. Hinton, and R. J. Williams, 1986, Learning internal representations by error propagation, in Parallel Distributed Processing, Vol. 1, Foundations, D. E. Rumelhart and J. L. McClelland, ed., MIT Press, Cambridge, MA, pp. 318-335.

[4] Widrow, B., 1962, Generalization and information storage in networks of ADELINE neurons, in Self-Organizing Systems, G. T. Yovitts, ed., Spartan Books, New York.