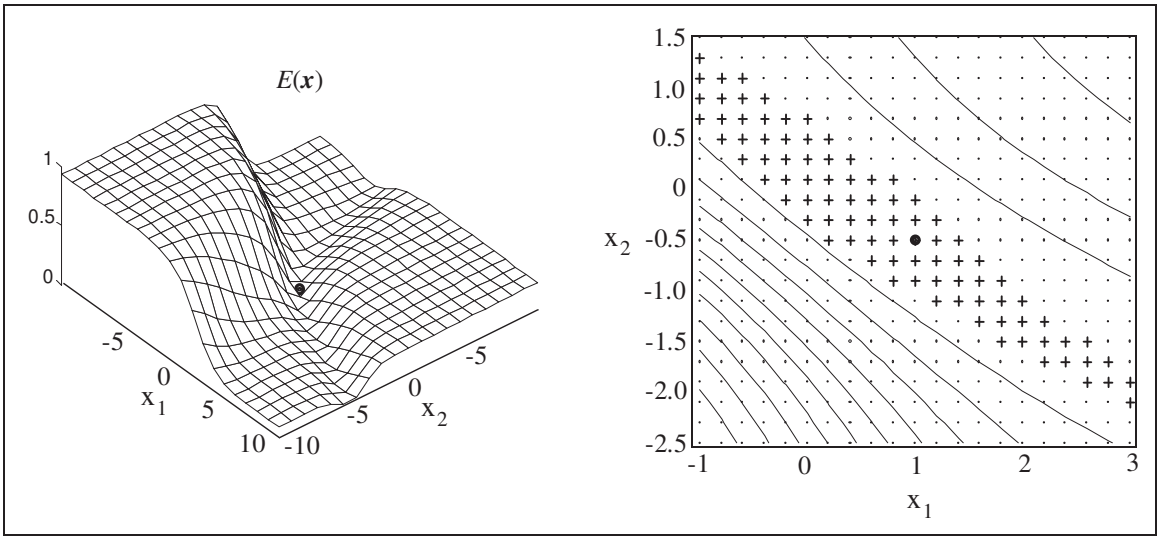

Figure 2 Error surface and definiteness of the 2-2-2 multilayer perceptron used in the example.

then the point is not the minimum any longer, but is still in the positive definite neighborhood of the new minimum. It can therefore be used as the new initial condition. If this process is repeated, the subsequent optimization processes 'chase' the moving minimum. An example of this class of problems is the inverse kinematics problem in robotics, where a sequence of joint angles is needed to guide the robot's end effector along a desired Cartesian trajectory. The minimum is initially known - one simply measures the joint angles. The robot arm can then be moved in small steps, while an optimization process continuously finds the best joint angles, never leaving the positive definite neighborhood of the (moving) minimum.

6 Conclusion

An efficient second-order optimization algorithm for the numerical inversion of nonlinear functions represented by feedforward neural networks was designed and its performance and limitations evaluated. As is well known, second-order information can dramatically reduces the number of iterations for convergence. It was demonstrated, that the algorithm developed herein, based on a neural network, also dramatically reduces the amount of computations required. Stringent conditions on the application context must be satisfied for a successful application of these algorithms. The method was demonstrated with an example, and a class of applicable problems was briefly characterized.

7 References

[1] J.W. Brewer, "Kronecker Products and Matrix Calculus in System Theory," *IEEE Transactions on Circuits and Systems*, Vol. 25, No. 9, pp. 772-781, September 1978.

[2] J. Dieudonné, *Foundations of Modern Analysis,* Academic Press, New York, 1960.

[3] M.R. Hestenes, *Conjugate Direction Methods in Optimization,* Springer-Verlag, New York, 1980.

[4] K. Hornik, M. Stinchcombe and H. White, "Multilayer Feedforward Networks are Universal Approximators," *Neural Networks,* Vol. 2, pp. 359-366, 1989.

[5] K. Mathia and R. Saeks, "Solving Nonlinear Equations Using Recurrent Neural Networks," *Proc. World Congress on Neural Networks 1995*, Washington D.C., Vol. 1, pp. 76-79, July 1995.

[6] M.F. Møller, "A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning," *Neural Networks*, Vol. 6, No. 4, pp. 525-533, 1993.

[7] B.A. Pearlmutter, "Fast Exact Multiplication by the Hessian," *Neural Computation*, Vol. 6, No. 1, pp. 147-160, January 1994.

[8] P.J. Werbos, "Beyond regression: New Tools for Prediction and Analysis in the Behavioral Sciences," Ph.D. Thesis, Harvard University, Cambridge, MA, 1974.

[9] P.J. Werbos, "Backpropagation: Past and Future," *Proc. IEEE Int. Conf. Neural Networks*, Vol. 1, pp. 343-353, San Diego, CA, 1988.

Since we constructed this example for demonstration purposes, we already know the desired fixed point $\boldsymbol{x}_f = (1, -0.5)$ and choose $\boldsymbol{x}_0 = (0, -0.2)$ as a convenient initial condition within the positive definite neighborhood of $\boldsymbol{x}_f$, as required by the theory. We choose $\|\boldsymbol{e}\|_\infty \leq 0.00001$ as the convergence criterion for the algorithms. From a mathematical perspective all three second-order methods compute precisely the same trajectory, but the efficiency of their implementations varies. Newton's method (Equation 8) is the most complex, because it not only computes the Hessian of the error function to be minimized, but also inverts it at each iteration. The standard CG algorithm (Figure 1) avoids the matrix inversion, but still needs the Hessian itself. The most efficient second-order technique of the three is the SCG algorithm which uses Equation 20 in the algorithm of Figure 1, to avoid both, computation and inversion of the Hessian. As shown in the previous section, Equation 12 takes the form of Equation 20 for the MLP in Equation 1. The values of Equation 21 were used in the experiment tabulated in Table 1. As is well known (and demonstrated here), the second-order techniques offer dramatic improvement in the number of iterations to converge over the first order (gradient descent) technique. The 'learning rate' was 1. It will be noticed there is similarly dramatic improvement in the amount of (total) computation required to achieve the solution. This was the objective of the algorithm we are presenting here. Although this example represents a small scale problem, we obtained similar results with larger networks.

For our example problem, we know the underlying equation, so we can determine the shape of the $E(\boldsymbol{x})$ surface (Figure 2). The minimum (our desired solution point) is shown as a black dot. The second figure 'zooms in' on the neighborhood of the minimum and shows the definiteness of the surface. Only a small fraction is positive definite (marked with '+'). All second order techniques require that the initial estimate (starting point) must be in this region. This is a limitation of the methodology.

Table 1: Algorithm Performances in the Example.

	SCG Algorithm	CG Algorithm	Newton's Method	Gradient Descent
Iterations	7	7	7	974
Flops	1924	2141	2582	53620
Fixed Point	[0.9999,-0.5000]	[0.9999,-0.5000]	[0.9999,-0.5000]	[0.9987,-0.4991]

5.3 Applicable Class of Problems

Important assumptions must be made when using second order optimization methods. First, the nonlinear function is (at least locally) invertible, i.e. input and output space have the same dimension. Second, a good estimate of the desired fixed point is required for choosing an initial condition in the positive definite neighborhood of that point. Third, the optimization process must not leave the positive definite portion of the error surface, otherwise the algorithm immediately can become unstable. This is a stringent constraint, since in the usual case only a small portion of the error surface is positive definite, as is shown in Figure 2 for our example. If we were to randomly select an initial condition within this 'mountainous' area, there is no reason to expect that it will be in that positive definite region.

Techniques have been proposed to overcome some of these difficulties [3][6], but any additional feature attached to the algorithms will cost computation time, whereas our goal is *efficient* problem solving for (possibly) time-critical applications. We rather impose constraints on the class of problems intended to be solved, and therefore assume *a priori* knowledge about the topology of the error function. This is reasonable near minima, where most functions behave like quadratics. Consider a scenario where the minimum is initially known (as in the example) and is used as the starting point. If the application context changes such that its corresponding error surface moves away from that point a sufficiently small step,

with $\boldsymbol{e}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{y}_d$. Minimizing E solves the nonlinear equation represented by the network, i.e. finds the network input $\boldsymbol{x}$ for a given desired output $\boldsymbol{y}_d$.

5.1 Derivation of the Algorithm

The application of Newton's method and the standard CG algorithm is straightforward if the Hessian of E is known. We apply the SCG algorithm for computing $\boldsymbol{Hp}$. With $\boldsymbol{g} = E'$ and $\boldsymbol{f}' = \partial \boldsymbol{f} / \partial \boldsymbol{x}$,

$$\boldsymbol{Hp} = R\{\boldsymbol{g}(\boldsymbol{x})\} = R\{\boldsymbol{e}^{\mathrm{T}}(\boldsymbol{x}) \cdot \boldsymbol{f}'(\boldsymbol{x})\} = R\{\boldsymbol{e}^{\mathrm{T}}(\boldsymbol{x})\} \cdot \boldsymbol{f}'(\boldsymbol{x}) + \boldsymbol{e}^{\mathrm{T}}(\boldsymbol{x}) \cdot R\{\boldsymbol{f}'(\boldsymbol{x})\} \quad . \tag{14}$$

The two $R\{.\}$ terms in Equation 14 are (with $\boldsymbol{a} = \boldsymbol{V}\boldsymbol{x}$, $\boldsymbol{a}_p = \boldsymbol{V}(\boldsymbol{x} + r\boldsymbol{p})$, $\mathbf{c} = \boldsymbol{V}\boldsymbol{p}$):

$$R\{\boldsymbol{e}^{\mathrm{T}}(\boldsymbol{x})\} = (\boldsymbol{W} \cdot \boldsymbol{\Sigma}'(\boldsymbol{a}) \cdot \boldsymbol{V} \cdot \boldsymbol{p})^{\mathrm{T}} , \tag{15}$$

$$R\{\boldsymbol{f}'(\boldsymbol{x})\} = R\{\boldsymbol{W} \cdot \boldsymbol{\Sigma}'(\boldsymbol{a}) \cdot \boldsymbol{V}\} = \boldsymbol{W} \cdot R\{\boldsymbol{\Sigma}'(\boldsymbol{a})\} \cdot \boldsymbol{V} \tag{16}$$

$$= \left[\frac{\partial \boldsymbol{a}_p^{\mathrm{T}} \boldsymbol{V}^{\mathrm{T}}}{\partial r}\right]_{r=0} \otimes \boldsymbol{I}_{row(\Sigma')} \cdot \boldsymbol{I}_{col(r)} \otimes \left[\frac{\partial \boldsymbol{\Sigma}'(\boldsymbol{a}_p)}{\partial \boldsymbol{a}_p}\right]_{r=0} \tag{17}$$

$$= \mathbf{c}^{\mathrm{T}} \otimes \boldsymbol{I}_q \cdot 1 \otimes \frac{\partial \boldsymbol{\Sigma}'(\boldsymbol{a})}{\partial \boldsymbol{a}} = \begin{bmatrix} \mathbf{c}^{\mathrm{T}} & & \\ & \cdots & \\ & & \mathbf{c}^{\mathrm{T}} \end{bmatrix} \cdot \begin{bmatrix} \frac{\partial \boldsymbol{\Sigma}'(\boldsymbol{a})}{\partial a_1} \\ \cdots \\ \frac{\partial \boldsymbol{\Sigma}'(\boldsymbol{a})}{\partial a_q} \end{bmatrix} \tag{18}$$

$$= \begin{bmatrix} c_1 \cdot \sigma''(a_1) & & \\ & \cdots & \\ & & c_q \cdot \sigma''(a_q) \end{bmatrix} = \Sigma_c''(\boldsymbol{a}) . \tag{19}$$

The operator $\oplus$ performs the Kronecker tensor product [1]. Using the above derivations gives

$$\boldsymbol{Hp} = [\boldsymbol{p}^{\mathrm{T}} \cdot \boldsymbol{V}^{\mathrm{T}} \cdot \boldsymbol{\Sigma}'(\boldsymbol{a}) \cdot \boldsymbol{W}^{\mathrm{T}} \cdot \boldsymbol{W} \cdot \boldsymbol{\Sigma}'(\boldsymbol{a}) \cdot \boldsymbol{V}] + [\boldsymbol{e}^{\mathrm{T}}(\boldsymbol{x}) \cdot \boldsymbol{W} \cdot \Sigma_c''(\boldsymbol{a}) \cdot \boldsymbol{V}]$$

, (20)

which can be numerically optimized for implementation. Thus we have derived a form of Equation 12 based on the specific structure of an MLP defined by Equation 1. Consequently, if such an MLP is trained to represent a given application problem we need only go into the trained MLP and read out the values of $\boldsymbol{V}, \boldsymbol{W}, \boldsymbol{b}$ for plugging into Equation 20 and then into Equation 9.

5.2 Example

We assume an application context in which the process to be solved is being represented by a MLP (Equation 1) which was trained using data from the process. We use the second order methods presented above (Newton, CG, SCG) to invert the nonlinear function which is represented by the neural network. In our example we use a two-variable system, and after training our (two input, two hidden PE, two output) MLP has the following parameter values and desired output $\boldsymbol{y}_d$:

$$\boldsymbol{V} = \begin{bmatrix} 1 & 0.4 \\ 0.6 & 1.5 \end{bmatrix}, \boldsymbol{W} = \begin{bmatrix} 1 & 0.5 \\ 0.4 & 1.0 \end{bmatrix}, \boldsymbol{b} = \begin{bmatrix} 0.3 \\ 0.5 \end{bmatrix}, \boldsymbol{y}_d = \begin{bmatrix} 1.0440 \\ 0.8867 \end{bmatrix}. \tag{21}$$

Choose $\mathbf{x}_1$ (column vector). Compute $\mathbf{H} = \mathrm{E}''(\mathbf{x}_1)$, $\mathbf{g}_1 = -\mathrm{E}'(\mathbf{x}_1)$. Set $\mathbf{p}_1 = \mathbf{g}_1$ (row vectors).
LOOP: i = 1..dim($\mathbf{x}$)

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i \mathbf{p}_i^{\mathrm{T}}, \quad \alpha_i = \mathbf{p}_i \mathbf{g}_i^{\mathrm{T}} / \mathbf{p}_i \mathbf{H} \mathbf{p}_i^{\mathrm{T}} \tag{9.a}$$

$$\mathbf{g}_{i+1} = \mathbf{g}_i - \alpha_i \mathbf{p}_i \mathbf{H} \tag{9.b}$$

$$\mathbf{p}_{i+1} = \mathbf{g}_{i+1} + \beta_i \mathbf{p}_i, \quad \beta_i = \mathbf{p}_i \mathbf{H} \mathbf{g}_{i+1}^{\mathrm{T}} / \mathbf{p}_i \mathbf{H} \mathbf{p}_i^{\mathrm{T}} \tag{9.c}$$

end

Figure 1 Conjugate gradient algorithm for minimizing a quadratic function.

One complete loop of the standard CG algorithm is equivalent to one Newton iteration and converges in n steps for *quadratic* functions (of course, the Hessian must be positive definite). The loop must be repeated for nonquadratic functions. Orthogonality of subsequent search directions is maintained with respect to two norms: one is the inner product $\boldsymbol{p}_i \boldsymbol{g}_i^{\mathrm{T}}$, the second is the 'weighted' inner product $\boldsymbol{p}_i \boldsymbol{H} \boldsymbol{p}_i^{\mathrm{T}}$. If computation of the Hessian is to be avoided, different estimation techniques are available for α_i and β_i, usually at the cost of slower convergence. The step size α_i is the solution to $min\{E(\boldsymbol{x}_i + \alpha_i \boldsymbol{p}_i)\}$ and can be found using a line search. For β_i well known estimates are available [3][6].

4.3 Scaled Conjugate Gradient Algorithm

The scaled conjugate gradient (SCG) algorithm circumvents the multiplication $\boldsymbol{H}\boldsymbol{p}_i^T$ and computes this vector directly, without computing the Hessian. An elegant way to compute the *exact* product $\boldsymbol{H}\boldsymbol{x}$ of a Hessian and an arbitrary vector $\boldsymbol{x}$ has been independently rediscovered for neural networks in [6],[7] and [9]. This method is used for the numerical inversion proposed in this paper, with the *exact* values for α_i and β_i. We begin with the first-order expansion of the error gradient,

$$\boldsymbol{g}(\boldsymbol{x} + \Delta\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{x}) + \Delta\boldsymbol{x}^{\mathrm{T}} \boldsymbol{H}. \tag{10}$$

Transposing Equation 10 and setting $\Delta\boldsymbol{x} = r\boldsymbol{p}$, with scalar r and vector $\boldsymbol{p}$, gives

$$r\boldsymbol{H}\boldsymbol{p} = \boldsymbol{g}(\boldsymbol{x} + r\boldsymbol{p}) - \boldsymbol{g}(\boldsymbol{x}). \tag{11}$$

The desired product $\boldsymbol{H}\boldsymbol{p}$ is obtained by dividing Equation 11 by r and taking the limit,

$$\boldsymbol{H}\boldsymbol{p} = \lim_{r \to 0} \frac{\boldsymbol{g}(\boldsymbol{x} + r\boldsymbol{p}) - \boldsymbol{g}(\boldsymbol{x})}{r} = \frac{\partial}{\partial r} [\boldsymbol{g}(\boldsymbol{x} + r\boldsymbol{p})]_{r=0} = R_p\{\boldsymbol{g}(\boldsymbol{x})\}, \tag{12}$$

which coincides with the definition of a derivative. The definition of the differential operator $R_p\{\boldsymbol{g}(\boldsymbol{x})\} = \boldsymbol{H}\boldsymbol{p}$ is due to [7]. The usual rules for differential operators apply. In this paper, we obtain the desired product by applying this operator to those equations which compute gradients. In the following we simplify the notation to $R\{.\}$.

5 Using Feedforward Neural Networks

The above second order optimization methods are applied to the numerical inversion of nonlinear functions represented by the MLP in Equation 1. The error function is the usual squared error,

$$E(\boldsymbol{x}) = 1/2 \cdot \boldsymbol{e}^{\mathrm{T}}(\boldsymbol{x}) \cdot \boldsymbol{e}(\boldsymbol{x}), \tag{13}$$

jugate gradient algorithm), 3) *both* computation and inversion of the Hessian are avoided (scaled conjugate gradient algorithm). We focus on 3), but also perform the computations in 1) and 2) for comparison purposes (see Section 5.2).

Second-order optimization techniques assume that a second-order Taylor series expansion is a sufficiently accurate approximation of the error function E about a point $\boldsymbol{x}$,

$$E(\boldsymbol{x}+\Delta\boldsymbol{x}) = E(\boldsymbol{x}) + \Delta E(\boldsymbol{x}) = E(\boldsymbol{x}) + E'(\boldsymbol{x})\Delta\boldsymbol{x} + \frac{1}{2}\Delta\boldsymbol{x}^{\mathrm{T}}E''(\boldsymbol{x})\Delta\boldsymbol{x}, \tag{5}$$

where E'' is the Hessian of E. This assumption is appropriate near minima, where higher order terms vanish. Of course we assume that E' and E'' exist and are continuous. The approximating function is minimized by solving

$$\frac{\partial}{\partial\Delta\boldsymbol{x}}\{\Delta E(\boldsymbol{x})\} = E'(\boldsymbol{x}) + \Delta\boldsymbol{x}^{\mathrm{T}}E''(\boldsymbol{x}) = \boldsymbol{0}^{\mathrm{T}}, \tag{6}$$

$$\Delta\boldsymbol{x}^{*} = -[E''(\boldsymbol{x})]^{-1}\cdot[E'(\boldsymbol{x})]^{\mathrm{T}} \quad , \tag{7}$$

where E'' must be *positive definite*. This is a stringent constraint as we will demonstrate below. Also note that the Hessian of continuous functions is symmetric, and that the gradient E' is a row vector [2]. For the remainder of this paper we write $\boldsymbol{g} = E'$ and $\boldsymbol{H} = E''$.

4.1 Newton's Method

The key calculation for all second-order optimization techniques based on Newton's methods is Equation 7. With $\Delta\boldsymbol{x} = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$ and our new notation, Newton's method is the iteration process

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \boldsymbol{H}_k^{-1}\cdot\boldsymbol{g}_k^{\mathrm{T}}. \tag{8}$$

with $\boldsymbol{H}_k = \boldsymbol{H}(\boldsymbol{x}_k)$, $\boldsymbol{g}_k = \boldsymbol{g}(\boldsymbol{x}_k)$. One iteration step minimizes the approximating function in Equation 5. Many iterations may be needed until the minimum of the approximating function is sufficiently close to the minimum of the original function. A drawback of Newton's method and its variants is the computation, storage, and inversion of the Hessian at each iteration step, which can be computationally expensive, in particular for high-dimensional systems. Quasi-Newton methods avoid these computations, for example the conjugate gradient algorithm.

4.2 Conjugate Gradient Method

The conjugate gradient (CG) method iteratively approximates the inverse Hessian matrix, while limiting the number of iteration steps to n by searching only along basis vectors of the n-dimensional search space. The vector from starting point $\boldsymbol{x}_1$ to the solution $\boldsymbol{x}^*$ is therefore some linear combination of n basis vectors $\boldsymbol{p}_i$ of length α_i,

$$\boldsymbol{x}^{*} - \boldsymbol{x}_1 = \sum_{i=1}^{n}\alpha_i\cdot\boldsymbol{p}_i. \tag{9}$$

The questions is: How can we obtain an appropriate set of basis vectors and the associated step sizes? Fortunately, if the gradient and the Hessian are known at the starting point, these entities can be computed while the algorithm progresses, based on the orthogonality of each new gradient $\boldsymbol{g}_{k+1}$ to the previous search direction $\boldsymbol{p}_k$ [3]. Which basis is being constructed depends on $\boldsymbol{x}_1$. The standard CG algorithm is shown in Figure 1.

hidden layer with q sigmoidal processing elements (PEs), and an output layer with n linear PEs. The network represents a nonlinear mapping $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$ with input vector $\boldsymbol{x}$ and output vector $\boldsymbol{y}$, defined by

$$\boldsymbol{f}: \Re^n \rightarrow \Re^n, \quad \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{W} \cdot \sigma(\boldsymbol{V}\boldsymbol{x} + \boldsymbol{b}) = \boldsymbol{W} \cdot \sigma(\boldsymbol{a}) \quad . \tag{1}$$

After training, the weight matrices of the hidden and output layers, $\boldsymbol{V} \in \Re^{q \times n}$ and $\boldsymbol{W} \in \Re^{n \times q}$, are fixed. The usual bias input $\boldsymbol{b}$ is applied to the hidden layer only. We define $\boldsymbol{a} = \boldsymbol{V}\boldsymbol{x} + \boldsymbol{b}$. The (column) vector-valued function $\sigma(\boldsymbol{a}) = [\sigma(a_1), \ldots, \sigma(a_q)]$ represents the sigmoid transfer functions of hidden PEs, where $\sigma(a_i) = 1/(1 + exp(-a_i))$.

2.2 Jacobian and Hessian

The Jacobian (matrix of first derivatives) and the Hessian (matrix of second derivatives) of the MLP in Equation 1 is needed for the derivation of the algorithms in Section 4. The Jacobian is

$$\frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial \boldsymbol{x}} = \boldsymbol{W} \cdot \frac{\partial \sigma(\boldsymbol{a})}{\partial \boldsymbol{a}} \cdot \boldsymbol{V} = \boldsymbol{W} \cdot \Sigma'(\boldsymbol{a}) \cdot \boldsymbol{V}, \tag{2}$$

where $\Sigma'(\boldsymbol{a})$ is a diagonal matrix with elements $\sigma'(a_i) = \sigma(a_i) \cdot (1 - \sigma(a_i))$, the well known form of a sigmoid's derivative. The Hessian's element in row i and column j is

$$\frac{\partial^2}{\partial \boldsymbol{x}_i \boldsymbol{x}_j} \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{W} \cdot (\Sigma''(\boldsymbol{a}) \cdot \boldsymbol{v}^i) \cdot \boldsymbol{v}^j, \tag{3}$$

where $\boldsymbol{v}^i$ and $\boldsymbol{v}^j$ are the i-th and j-th column vectors in $\boldsymbol{V}$, respectively. $\Sigma''(\boldsymbol{a})$ is a diagonal matrix with elements $\sigma''(a_i) = \sigma(a_i) - 3\sigma^2(a_i) + 2\sigma^3(a_i)$. It is clear that the computation of the Hessian (second-order information!) even for this relatively simple network architecture is considerably complex.

3 First-Order Optimization

A popular first-order technique in the context of artificial neural networks is error backpropagation [8], a training algorithm for optimizing the connection weights of a feedforward ANN for a given problem. 'Backprop' can be viewed as an unconstrained nonlinear optimization scheme which combines the classical gradient descent and backsubstitution methods. It is based on the assumption that a first-order approximation of E about some point $\boldsymbol{x}$ is sufficiently accurate, thus $E(\boldsymbol{x} + \Delta\boldsymbol{x}) = E(\boldsymbol{x}) + E'(\boldsymbol{x})\Delta\boldsymbol{x}$, where $E' = \partial E / \partial \boldsymbol{x}$ is the error gradient. Backprop iteratively minimizes and updates the approximating function. A potential drawback of first-order methods is poor convergence, which mainly corresponds to a poor first-order approximation. The backprop weight update rule therefore often includes the 'momentum term',

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta\boldsymbol{x}_k + \Delta\boldsymbol{x}_{k-1} = \boldsymbol{x}_k + \eta E'(\boldsymbol{x}) + \mu(\boldsymbol{x}_k - \boldsymbol{x}_{k-1}) \quad , \tag{4}$$

where $\boldsymbol{x}$ denotes a vector of connection weights and the scalar η the learning rate. The momentum term $\Delta\boldsymbol{x}_{k-1}$, with the momentum constant μ, has been suggested as a (not always successful) *ad hoc* attempt to include *second-order information* in order to increase convergence speed. Also note that training algorithms like backprop search the *weight space* of a network, whereas we are concerned with searching the *input space* (see also [5]).

4 Second-Order Optimization

In this section we review second-order optimization techniques of decreasing computational complexity, decreasing in the sense that - depending on the algorithm - the Hessian of a scalar, multi-variate error function is 1) computed *and* inverted (Newton's method), 2) *only* computed and its inversion avoided (con-

World Congress on Neural Networks 1996, San Diego, California.

Efficient Numerical Inversion Using Multilayer Feedforward Neural Networks

George Lendaris
Professor for Systems Science and Electrical Engineering
Portland State University
Portland, Oregon
lendaris@sysc.pdx.edu

Karl Mathia
Senior Research Engineer
Accurate Automation Corporation
Chattanooga, Tennessee
kmathia@accurate-automation.com

Abstract

An efficient second-order optimization algorithm for the numerical inversion of nonlinear functions using feedforward neural networks is presented. After a function has been successfully learned by a neural network, the (nonlinear) equation is solved using the neural network representation, in effect, performing a numerical inversion process. As is well known, second-order information can dramatically improve the convergence of numerical methods. It is demonstrated, that the algorithm developed herein, based on a neural network, also dramatically reduces the amount of computations required, hence yielding a more efficient numerical inversion process. Certain constraints must also be considered. We discuss these issues, present an example and briefly characterize an applicable class of problems.

1 Introduction

An efficient second-order optimization algorithm for the numerical inversion of nonlinear functions using feedforward neural networks is presented. After a function has been successfully learned by a neural network, the (nonlinear) equation is solved using the neural network representation, in effect, performing a numerical inversion process. For example, this is useful when a function is given only in terms of sample points. A common scenario for solving nonlinear and high-dimensional functions is to define and solve an optimization problem. We follow this strategy and define a scalar, multi-variate error function $E(\boldsymbol{x})$ and search for the optimal solution vector $\boldsymbol{x}^*$, such that $E(\boldsymbol{x}^*) = min\{E(\boldsymbol{x})\}$. Many numerical optimization schemes are available for this purpose. A technique is called of 'order n' if it uses a n-th order Taylor series expansion to approximate E, and then iteratively minimizes and updates the approximating function instead of the original. Incorporating higher order information, i.e. up to the n-th derivative of E, provides a better approximation and therefore can dramatically improve convergence. Unfortunately, higher order terms also can make the numerical process unreasonably complex. Many optimization schemes therefore approximate not only the original function, but also its derivatives. We design and evaluate an efficient algorithm for a class of feedforward neural networks. Some important limitations of the algorithm are also discussed.

In the following sections we define the architecture of a feedforward multilayer perceptron (MLP) which represents the family of neural networks under consideration. Second-order techniques of decreasing computational complexity are reviewed, 'decreasing' in the sense that - depending on the algorithm - derivatives and matrix inversions are computed or approximated. We apply the algorithms to the networks previously defined, and demonstrate the techniques (and its limitations) with an example. We conclude with a discussion and characterization of an applicable class of problems.

2 Feedforward Neural Networks

The class of neural networks used as an example in this paper is the common MLP, although the algorithm is applicable to similar feedforward architectures with only minor modifications.

2.1 Universal Approximator Network

The feedforward neural network defined in the universal approximator theorem [4], the MLP, is used as an example network for design and evaluation of the algorithm. The network has n input nodes, one