Training Feedforward Networks

Martin T. Hagan and Mo

***Abstract*— The Marquardt algorithm for nonlinear least squares is presented and is incorporated into the backpropagation algorithm for training feedforward neural networks. The algorithm is tested on several function approximation problems, and is compared with a conjugate gradient algorithm and a variable learning rate algorithm. It is found that the Marquardt algorithm is much more efficient than either of the other techniques when the network contains no more than a few hundred weights.**

I. Introduction

SINCE the backpropagation learning algorithm [1] was first popularized, there has been considerable research

I. Introduction

SINCE the backpropagation learning algorithm [1] was first popularized, there has been considerable research on methods to accelerate the convergence of the algorithm. This research falls roughly into two categories. The first category involves the development of ad hoc techniques (e.g., [2]–[5]). These techniques include such ideas as varying the learning rate, using momentum and rescaling variables. Another category of research has focused on standard numerical optimization techniques (e.g., [6]–[9]).

The most popular approaches from the second category have used conjugate gradient or quasi-Newton (secant) methods. The quasi-Newton methods are considered to be more efficient, but their storage and computational requirements go up as the square of the size of the network. There have been some limited memory quasi-Newton (one step secant) algorithms that speed up convergence while limiting memory requirements [8,10]. If exact line searches are used, the one step secant methods produce conjugate directions.

speed up convergence while limiting memory requirements [8,10]. If exact line searches are used, the one step secant methods produce conjugate directions.

Another area of numerical optimization that has been applied to neural networks is nonlinear least squares [11]–[13]. The more general optimization methods were designed to work effectively on all sufficiently smooth objective functions. However, when the form of the objective function is known it is often possible to design more efficient algorithms. One particular form of objective function that is of interest for neural networks is a sum of squares of other nonlinear functions. The minimization of objective functions of this type is called nonlinear least squares.

Most of the applications of nonlinear least squares to neural networks have concentrated on sequential implementations, where the weights are updated after each presentation of an input/output pair. This technique is useful when on-line adap-

$$\sum_{t=1}^{N} \left(y[t] - \bar{F}(x[t]|\theta) \right)^2$$

$$e[t] = y[t] - \bar{F}(x[t]|\theta)$$

Another area of numerical optimization that has been applied to neural networks is nonlinear least squares [11]–[13]. The more general optimization methods were designed to work effectively on all sufficiently smooth objective functions. However, when the form of the objective function is known it is often possible to design more efficient algorithms. One particular form of objective function that is of interest for neural networks is a sum of squares of other nonlinear functions. The minimization of objective functions of this type is called nonlinear least squares.

Most of the applications of nonlinear least squares to neural networks have concentrated on sequential implementations, where the weights are updated after each presentation of an input/output pair. This technique is useful when on-line adaptation is needed, but it requires that several approximations be made to the standard algorithms. The standard algorithms are performed in batch mode, where the weights are only updated after a complete sweep through the training set.

three-layer netwo

The net input t

$$n^{k+1}(i)$$

The output of un

For an M layer n
are given by

$$\underline{a}^{k+}$$

The task of the ne
ified set of input–

1045–9227/94$04.00 © 1994 IEEE

ks with the Marquardt Algorithm

Mohammad B. Menhaj

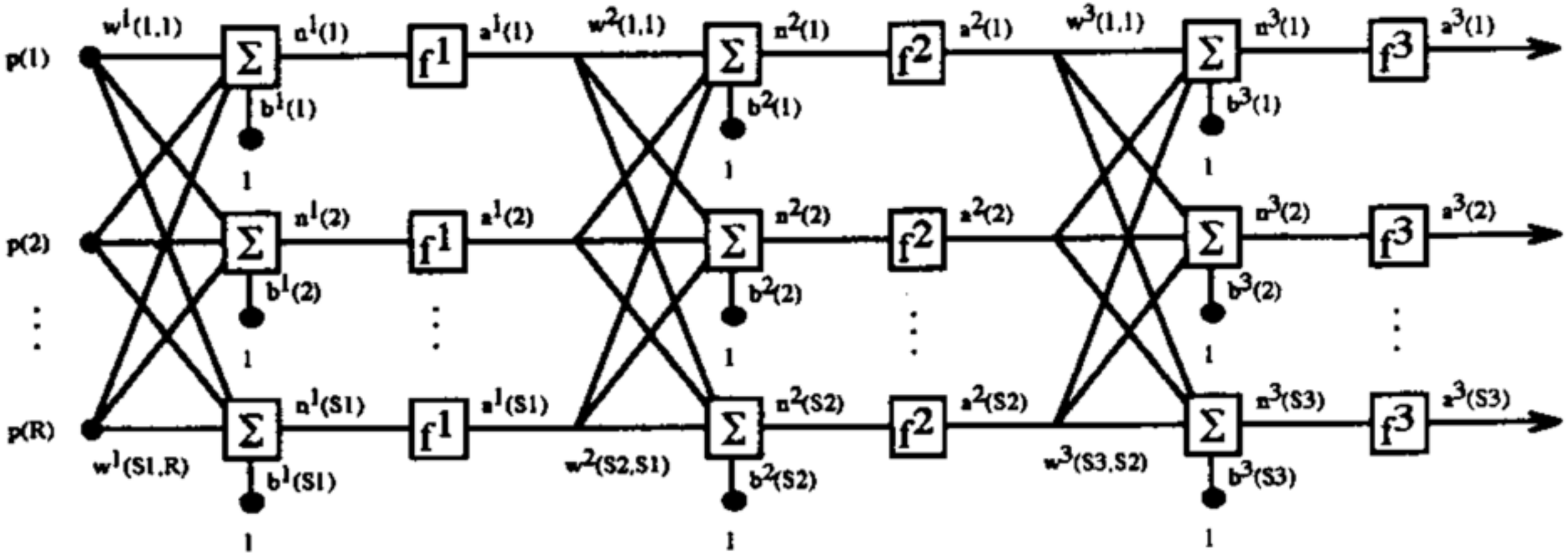


Fig. 1. Three-layer feedforward network.

This paper presents the application of a nonlinear least squares algorithm to the batch training of multi-layer percep-

This paper presents the application of a nonlinear least squares algorithm to the batch training of multi-layer perceptrons. For very large networks the memory requirements of the algorithm make it impractical for most current machines (as is the case for the quasi-Newton methods). However, for networks with a few hundred weights the algorithm is very efficient when compared with conjugate gradient techniques. Section II briefly presents the basic backpropagation algorithm. The main purpose of this section is to introduce notation and concepts which are needed to describe the Marquardt algorithm. The Marquardt algorithm is then presented in Section III. In Section IV the Marquardt algorithm is compared with the conjugate gradient algorithm and with a variable learning rate variation of backpropagation. Section V contains a summary and conclusions.

II. Backpropagation Algorithm

Consider a multilayer feedforward network, such as the three-layer network of Fig. 1.

The net input to unit i in layer $k+1$ is

$$n^{k+1}(i) = \sum_{j=1}^{Sk} w^{k+1}(i,j)a^k(j) + b^{k+1}(i). \tag{1}$$

The output of unit i will be

$$a^{k+1}(i) = f^{k+1}\left(n^{k+1}(i)\right). \tag{2}$$

For an M layer network the system equations in matrix form are given by

$$\underline{a}^0 = \underline{p} \tag{3}$$

$$\underline{a}^{k+1} = \underline{f}^{k+1}\left(W^{k+1}\underline{a}^k + \underline{b}^{k+1}\right),$$

$$k = 0, 1, \cdots, M-1. \tag{4}$$

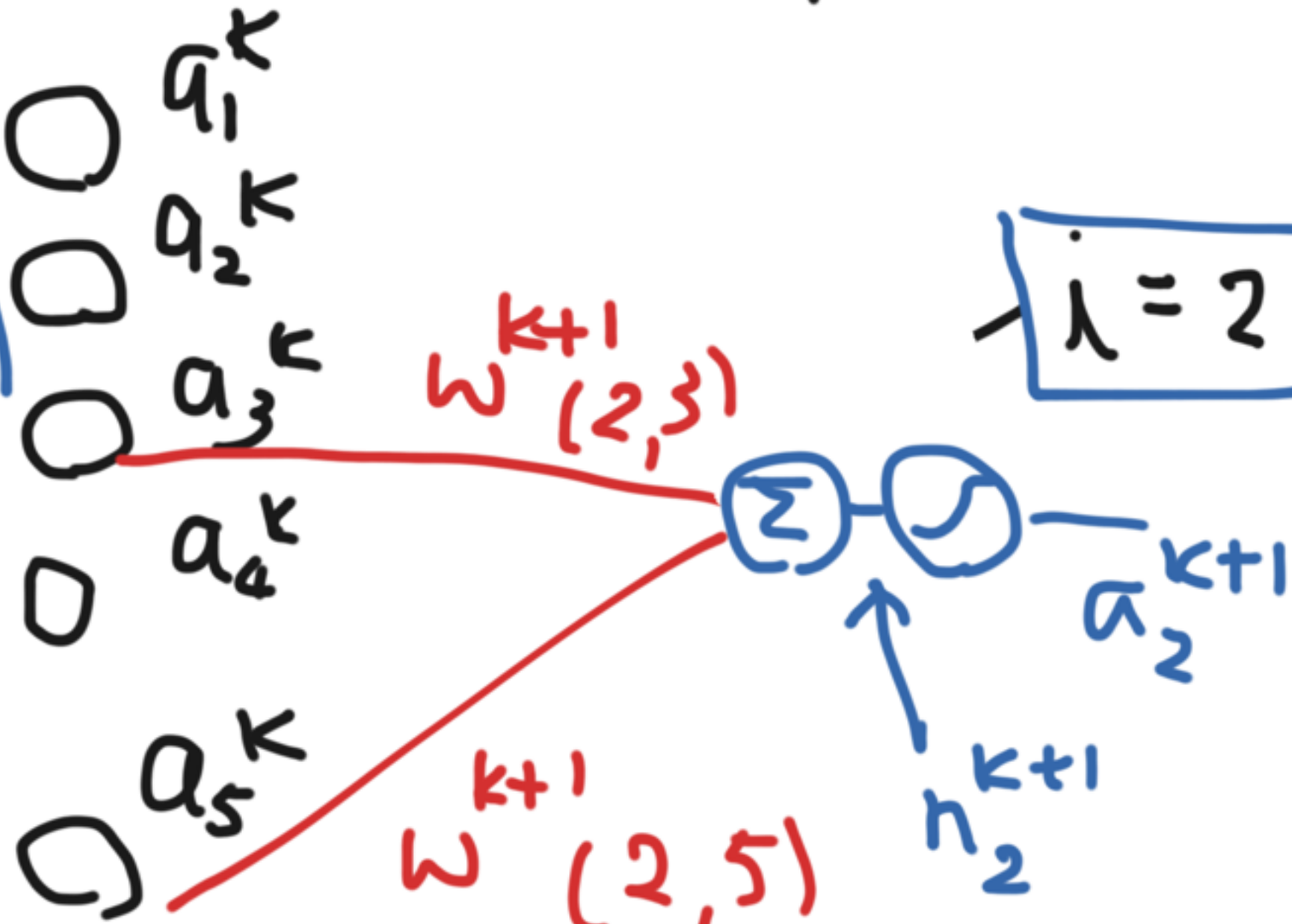
The task of the network is to learn associations between a specified set of input–output pairs $\{(\underline{p}_1, \underline{t}_1), (\underline{p}_2, \underline{t}_2), \cdots, (\underline{p}_Q, \underline{t}_Q)\}$.

$S_k = 5$

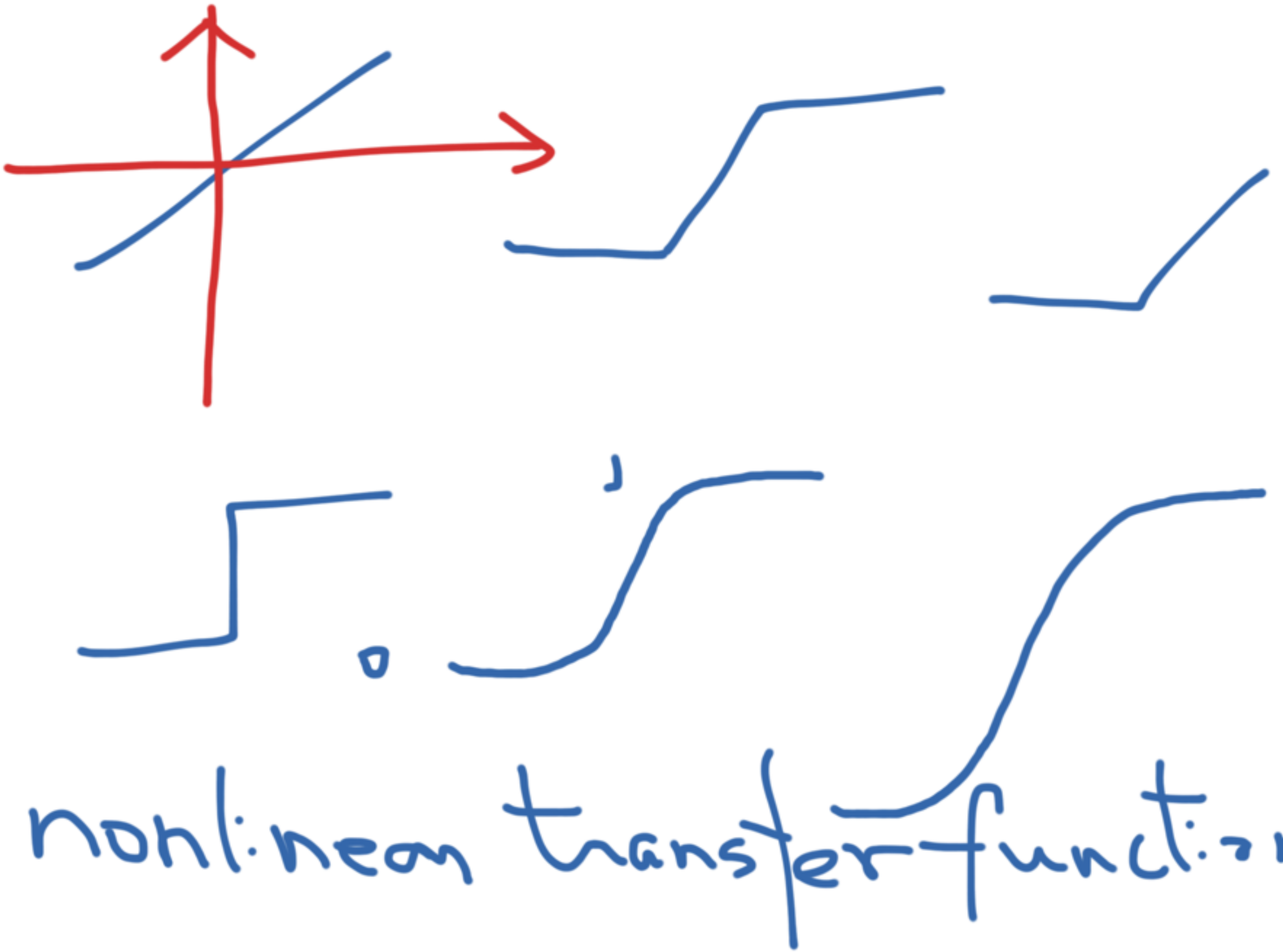
$S_{k+1} = 3$

$$n_i^{k+1} = \sum_{j=1}^{S_k} a_j^k \, w(i,j) + b_j^{k+1}$$

$\lambda = 2$



$$n_2^{k+1} = a_1^k w^{k+1}(2,1) + a_2^k w^{k+1}(2,2) + \cdots + a_5^k w^{k+1}(2,5)$$


nonlinear transfer function

The performance index for the network is

$$V = \frac{1}{2}\sum_{q=1}^{Q}\left(\underline{t}_q - \underline{a}_q^M\right)^T\left(\underline{t}_q - \underline{a}_q^M\right) = \frac{1}{2}\sum_{q=1}^{Q}\underline{e}_q^T\,\underline{e}_q \qquad (5)$$

where $\underline{a}_q^M$ is the output of the network when the qth input, $\underline{p}_q$, is presented, and $\underline{e}_q = \underline{t}_q - \underline{a}_q^M$ is the error for the qth input. For the standard backpropagation algorithm we use an approximate steepest descent rule. The performance index is approximated by

$$\hat{V} = \frac{1}{2}\underline{e}_q^T\,\underline{e}_q \qquad (6)$$

where the total sum of squares is replaced by the squared errors for a single input/output pair. The approximate steepest (gradient) descent algorithm is then

$$\Delta w^k(i,j) = -\alpha \frac{\partial \hat{V}}{\partial w^k(i,j)} \tag{7}$$

$$\Delta b^k(i) = -\alpha \frac{\partial \hat{V}}{\partial b^k(i)} \tag{8}$$

where α is the learning rate. Define

$$\delta^k(i) \equiv \frac{\partial \hat{V}}{\partial n^k(i)} \tag{9}$$

as the sensitivity of the performance index to changes in the net input of unit i in layer k. Now it can be shown, using (1), (6), and (9), that

$$\frac{\partial \hat{V}}{\partial\, w^k(i,j)} = \frac{\partial \hat{V}}{\partial\, n^k(i)}\,\frac{\partial n^k(i)}{\partial\, w^k(i,j)} = \delta^k(i)a^{k-1}(j) \qquad (10)$$

$$\frac{\partial \hat{V}}{\partial\, b^k(i)} = \frac{\partial \hat{V}}{\partial\, n^k(i)}\,\frac{\partial n^k(i)}{\partial\, b^k(i)} = \delta^k(i). \qquad (11)$$

$$\underline{\delta}^k = \dot{F}^k\left(\underline{n}^k\right) W^{k+1^\top} \underline{\delta}^{k+1} \tag{12}$$

where

$$\dot{F}^k(\underline{n}^k) = \begin{bmatrix} \dot{f}^k(n^k(1)) & 0 & \cdots & 0 \\ 0 & \dot{f}^k(n^k(2)) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \dot{f}^k(n^k(Sk)) \end{bmatrix} \tag{13}$$

and

$$\dot{f}^k(n) = \frac{df^k(n)}{dn}. \tag{14}$$

The overall learning algorithm now proceeds as follows; first, propagate the input forward using (3)–(4); next, propagate the sensitivities back using (15) and (12); and finally, update the weights and offsets using (7), (8), (10), and (11).

III. Marquardt-Levenberg Modification

While backpropagation is a steepest descent algorithm, the Marquardt-Levenberg algorithm [14] is an approximation to Newton's method. Suppose that we have a function $V(\underline{x})$ which we want to minimize with respect to the parameter vector $\underline{x}$, then Newton's method would be

$$\Delta \underline{x} = -[\nabla^2 V(\underline{x})]^{-1} \nabla V(\underline{x}) \tag{16}$$

where $\nabla^2 V(\underline{x})$ is the Hessian matrix and $\nabla V(\underline{x})$ is the gradient. If we assume that $V(\underline{x})$ is a sum of squares function

where $\nabla^2 V(\underline{x})$ is the Hessian matrix and $\nabla V(\underline{x})$ is the gradient. If we assume that $V(\underline{x})$ is a sum of squares function

$$V(\underline{x}) = \sum_{i=1}^{N} e_i^2(\underline{x}) \tag{17}$$

then it can be shown that

$$\nabla V(\underline{x}) = J^T(\underline{x})\underline{e}(\underline{x}) \tag{18}$$

$$\nabla^2 V(\underline{x}) = J^T(\underline{x})J(\underline{x}) + S(\underline{x}) \tag{19}$$

$$S(\underline{x}) = \sum_{i=1}^{N} e_i(\underline{x}) \nabla^2 e_i(\underline{x}). \tag{21}$$

For the Gauss-Newton method it is assumed that $S(\underline{x}) \approx 0$, and the update (16) becomes

$$\Delta \underline{x} = [J^T(\underline{x}) J(\underline{x})]^{-1} J^T(\underline{x}) \underline{e}(\underline{x}). \tag{22}$$

The Marquardt-Levenberg modification to the Gauss-Newton method is

$$\Delta \underline{x} = [J^T(\underline{x})J(\underline{x}) + \mu I]^{-1} J^T(\underline{x})\underline{e}(\underline{x}). \tag{23}$$

The parameter μ is multiplied by some factor (β) whenever a step would result in an increased $V(\underline{x})$. When a step reduces $V(\underline{x})$, μ is divided by β. (In Section IV we used μ =0.01 as a starting point, with β=10.) Notice that when μ is large the algorithm becomes steepest descent (with step $1/\mu$), while for small μ the algorithm becomes Gauss-Newton. The Marquardt-Levenberg algorithm can be considered a trust-region modification to Gauss-Newton [8].

The key step in this algorithm is the computation of the Jacobian matrix. For the neural network mapping problem the terms in the Jacobian matrix can be computed by a simple modification to the backpropagation algorithm. The performance index for the mapping problem is given by (5). It is easy to see that this is equivalent in form to (17), where $\underline{x} = [w^1(1,1)w^1(1,2)\cdots w^1(S1,R)b^1(1)\cdots b^1(S1)w^2(1,1)\cdots b^M(SM)]^T$, and $N = Q \times SM$. Standard backpropagation calculates terms like

$$\frac{\partial \hat{V}}{\partial w^k(i,j)} = \frac{\partial \sum_{m=1}^{SM} e_q^2(m)}{\partial w^k(i,j)}. \tag{24}$$

For the elements of the Jacobian matrix that are needed for the Marquardt algorithm we need to calculate terms like

$$\frac{\partial e_q(m)}{\partial w^k(i,j)} \tag{25}$$

These terms can be calculated using the standard backpropagation algorithm with one modification at the final layer

$$\Delta^M = -\dot{F}^M(\underline{n}^M). \tag{26}$$

Note that each column of the matrix in (26) is a sensitivity vector which must be backpropagated through the network to produce one row of the Jacobian.

The Marquardt modification to the backpropagation algorithm

1) Present all inputs to the network and compute the corresponding network outputs (using (3) and (4)), and errors ($\underline{e}_q = \underline{t}_q - \underline{a}_q^M$). Compute the sum of squares of errors over all inputs ($V(\underline{x})$).
2) Compute the Jacobian matrix (using (26), (12), (10), (11), and (20)).
3) Solve (23) to obtain $\Delta\underline{x}$. (For the results shown in the next section Cholesky factorization was used to solve this equation.)

4) Recompute the sum of squares of errors using $\underline{x} + \Delta\underline{x}$. If this new sum of squares is smaller than that computed in step 1, then reduce μ by β, let $\underline{x} = \underline{x} + \Delta\underline{x}$, and go back to step 1. If the sum of squares is not reduced, then increase μ by β and go back to step 3.
5) The algorithm is assumed to have converged when the norm of the gradient ((18)) is less than some predetermined value, or when the sum of squares has been reduced to some error goal.

For the first test problem a 1–15–1 network, with a hidden layer of sigmoid nonlinearities and a linear output layer, was trained to approximate the sinusoidal function

$$y = 1/2 + 1/4 \sin(3\pi x)$$

using MBP, CGBP and VLBP. Fig. 2 displays the training curves for the three methods. The training set consisted of 40 input/output pairs, where the input values were scattered in the interval $[-1,1]$; and the network was trained until the sum of squares of the errors was less than the error goal of 0.02. The curves shown in Fig. 2 are an average over 5 different initial weights. The initial weights are random, but are normalized using the method of Nguyen and Widrow [16].

The fourth test problem is a four input-single output function

$$y = \sin(2\pi x_1)x_2^2 x_3^3 x_4^4 e^{-(x_1+x_2+x_3+x_4)}. \tag{27}$$

For this example a 4–50–1 network (301 parameters) is trained to approximate (27), where the input values were scattered in the interval $[-1,1]$; and the network was trained until the sum of squares of the errors (over 400 input/output sets) was less that the error goal of 0.5. Fig. 7 displays the average learning curves (3 different initial weights), and line 4 of Table I summarizes the average results. The CGBP algorithm takes approximately four times as many flops as MBP (VLBP was not applied to this problem because of excessive training time)

We noted that the difference between the performances of MBP and CGBP became more pronounced as higher precision approximations were required. This effect is illustrated in Fig. 9. In this example a 1–10–1 network is trained to approximate the sine wave of (28). The sample size is held constant at 100 points, but the mean square error goal is halved in steps from 2×10^{-4} to 1.6×10^{-6}. Fig. 9 displays the number of flops required for convergence, as a function of the error goal, for both MBP and CGBP. The curves represent an average over 10 different initial conditions. With an error goal of 2×10^{-4} MBP is 16 times faster than CGBP. This ratio increases as the error goal is reduced; when the error goal is 1.6×10^{-6}, MBP is 136 times faster than CGBP.

Many numerical optimization techniques have been successfully used to speed up convergence of the backpropagation learning algorithm. This paper presented a standard nonlinear least squares optimization algorithm, and showed how to incorporate it into the backpropagation algorithm. The Marquardt algorithm was tested on several function approximation problems, and it was compared with the conjugate gradient algorithm and with variable learning rate backpropagation. The results indicate that the Marquardt algorithm is very efficient when training networks which have up to a few hundred weights. Although the computational requirements are much higher for each iteration of the Marquardt algorithm, this is more than made up for by the increased efficiency. This is especially true when high precision is required.

```
%% Train Neural Network Using |trainlm| Train Function
% This example shows how to train a neural network using the |trainlm|
% train function.
%
% Here a neural network is trained to predict body fat percentages.

[x, t] = bodyfat_dataset;
net = feedforwardnet(10, 'trainlm');
net = train(net, x, t);
y = net(x);
```

```
dim=4;
N=800;
x=rand(N,dim)*2-1;
y=sin(2*pi*x(:,1)).*x(:,2).^2.*x(:,3).^3.*x(:,4).^4.*exp(-sum(x,2));
```

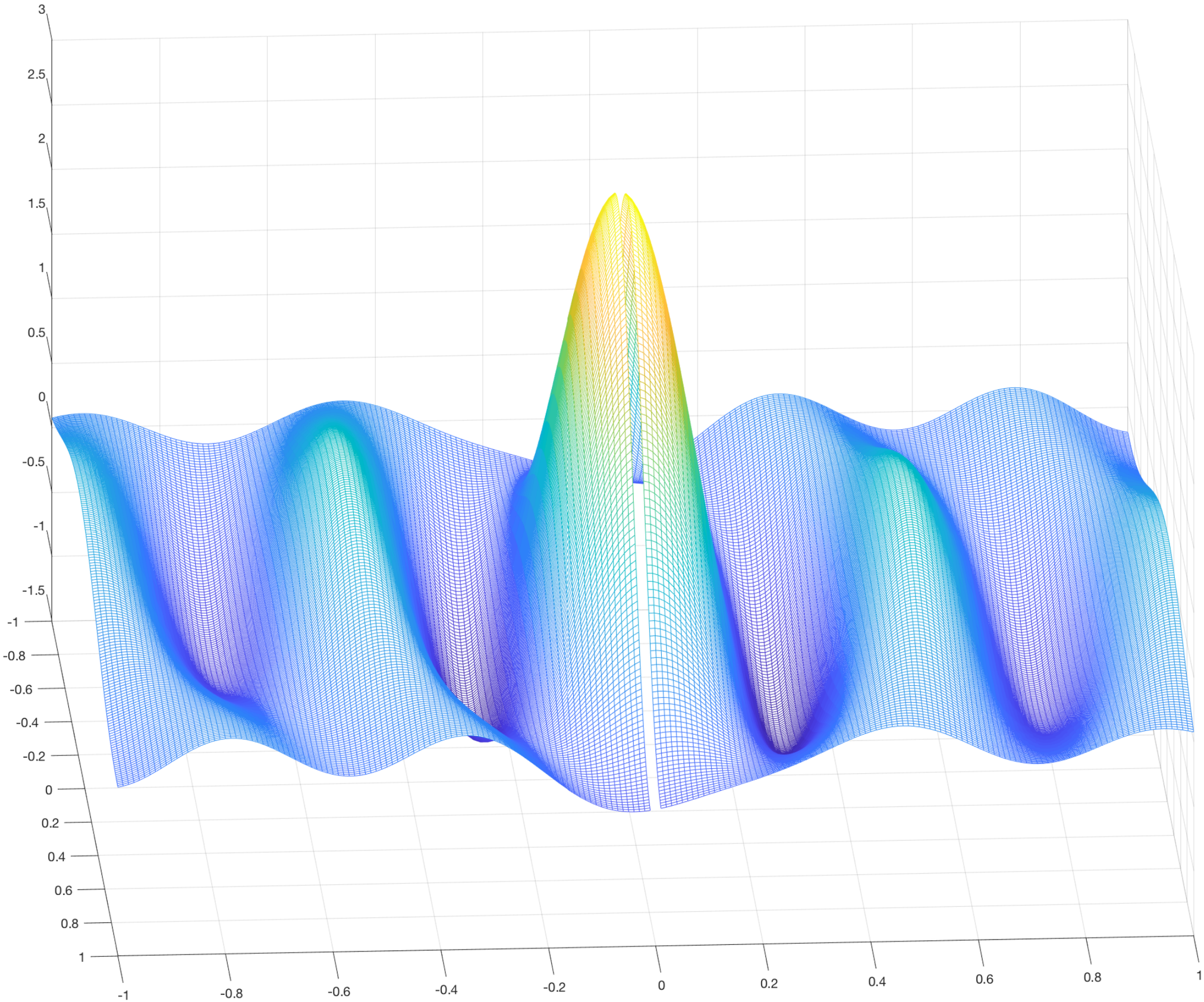
$$\mathbf{u}_1 = \begin{bmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{2} \end{bmatrix} \quad \text{and} \quad \mathbf{u}_2 = \begin{bmatrix} \frac{1}{2} \\ -\frac{\sqrt{3}}{2} \end{bmatrix}.$$

Denoting

$$\boldsymbol{\xi}_1 = \tfrac{2}{3}\mathbf{u}_1, \quad \boldsymbol{\xi}_2 = \tfrac{2}{3}\mathbf{u}_2, \quad \boldsymbol{\xi}_3 = -\tfrac{2}{3}(\mathbf{u}_1 + \mathbf{u}_2), \quad \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix},$$

one can derive[8] the sinc function for this hexagonal lattice as:

$$\begin{aligned} \operatorname{sinc}_{\mathrm{H}}(\mathbf{x}) = \tfrac{1}{3}\big(&\cos(\pi\boldsymbol{\xi}_1 \cdot \mathbf{x}) \operatorname{sinc}(\boldsymbol{\xi}_2 \cdot \mathbf{x}) \operatorname{sinc}(\boldsymbol{\xi}_3 \cdot \mathbf{x}) \\ &+ \cos(\pi\boldsymbol{\xi}_2 \cdot \mathbf{x}) \operatorname{sinc}(\boldsymbol{\xi}_3 \cdot \mathbf{x}) \operatorname{sinc}(\boldsymbol{\xi}_1 \cdot \mathbf{x}) \\ &+ \cos(\pi\boldsymbol{\xi}_3 \cdot \mathbf{x}) \operatorname{sinc}(\boldsymbol{\xi}_1 \cdot \mathbf{x}) \operatorname{sinc}(\boldsymbol{\xi}_2 \cdot \mathbf{x})\big) \end{aligned}$$


3
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1

https://matlab.mathworks.com/users/jmwu@mail.ndhu.edu.tw/Published/sinch_plot.html

https://matlab.mathworks.com/users/jmwu@mail.ndhu.edu.tw/Published/demo_hagan.html

where $J(x)$ is the Jacobian matrix

$$J(\underline{x}) = \begin{bmatrix} \frac{\partial e_1(\underline{x})}{\partial x_1} & \frac{\partial e_1(\underline{x})}{\partial x_2} & \cdots & \frac{\partial e_1(\underline{x})}{\partial x_n} \\ \frac{\partial e_2(\underline{x})}{\partial x_1} & \frac{\partial e_2(\underline{x})}{\partial x_2} & \cdots & \frac{\partial e_2(\underline{x})}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e_N(\underline{x})}{\partial x_1} & \frac{\partial e_N(\underline{x})}{\partial x_2} & \cdots & \frac{\partial e_N(\underline{x})}{\partial x_n} \end{bmatrix} \tag{20}$$

and

$$S(\underline{x}) = \sum_{i=1}^{N} e_i(\underline{x}) \nabla^2 e_i(\underline{x}). \tag{21}$$

Small steps and giant leaps: Minimal Newton solvers for Deep Learning

João F. Henriques **Sebastien Ehrhardt** **Samuel Albanie** **Andrea Vedaldi**
Visual Geometry Group, University of Oxford
{joao,hyenal,albanie,vedaldi}@robots.ox.ac.uk

click to source

Abstract

We propose a fast second-order method that can be used as a drop-in replacement for current deep learning solvers. Compared to stochastic gradient descent (SGD), it only requires two additional forward-mode automatic differentiation operations per iteration, which has a computational cost comparable to two standard forward passes and is easy to implement. Our method addresses long-standing issues with current second-order solvers, which invert an approximate Hessian matrix every iteration exactly or by conjugate-gradient methods, a procedure that is both costly and sensitive to noise. Instead, we propose to keep a single estimate of the gradient projected by the inverse Hessian matrix, and update it once per iteration. This estimate has the same size and is similar to the momentum variable that is commonly used in SGD. No estimate of the Hessian is maintained. We first validate our method, called CURVEBALL, on small problems with known closed-form solutions (noisy Rosenbrock function and degenerate 2-layer linear networks), where current deep learning solvers seem to struggle. We then train several large models on CIFAR and ImageNet, including ResNet and VGG-f networks, where we demonstrate faster convergence with no hyperparameter tuning. Code is available.