Efficient Training of Feed-Forward Neural Networks

Ph.D. Thesis
by

Martin Møller

Computer Science Department
Aarhus University
DK-8000 Århus, Denmark

November 21, 1997

Preface

Since the discovery of the back-propagation method, many modified and new algorithms have been proposed for training of feed-forward neural networks. The problem with slow convergence rate has, however, not been solved when the training is on large scale problems. There is still a need for more efficient algorithms. This Ph.D. thesis describes different approaches to improve convergence. The main results of the thesis is the development of the Scaled Conjugate Gradient Algorithm and the stochastic version of this algorithm. Other important results are the development of methods that can derive and use Hessian information in an efficient way. The main part of this thesis is the 5 papers presented in appendices A-E. Chapters 1-6 give an overview of learning in feed-forward neural networks, put these papers in perspective and present the most important results. The conclusion of this thesis is:

- Conjugate gradient algorithms are very suitable for training of feed-forward networks.
- Use of second order information by calculations on the Hessian matrix can be used to improve convergence.

I would like to thank Brian Mayoh for being a very inspiring advisor. Also many thanks to Ole Østerby who has been a great help on many technical issues. During my visit to Carnegie Mellon University, Pittsburgh, I received inspiration and advice from many people, which I can not all thank here. A great thanks, however, to my advisor at CMU, Scott Fahlman.

Thank you to the Royal Danish Research Council and the Carlsberg Foundation, who made this research possible by providing financial support.

Martin Møller

DAIMI, Århus Universitet
Juli 1993

Contents

Chapter 1

Resume in danish

Formålet med denne Ph.D. afhandling er udvikling af metoder til effektiv træning af feedforward neurale netværk. Afhandlingen består af 5 selvstændige artikler (appendix A-E) samt en oversigtsartikel.

1.1 Oversigtsartikel

Formålet med artiklen er at sætte resultaterne beskrevet i appendix A-E ind i en større sammenhæng og beskrive relevant litteratur.

1.1.1 Kapitel 2

Kapitlet præsenterer den notation, som er benyttet gennem hele afhandlingen.

1.1.2 Kapitel 3

Kapitel 3 er en gennemgang af de mest gængse og effektive træningsmetoder i litteraturen. I dette kapitel præsenteres resultaterne fra appendix A og B, som omhandler udvikling af en special designet konjugeret gradient algoritme og en stokastisk version af denne.

1.1.3 Kapitel 4

Kapitel 4 præsenterer nye ideer og metoder til effektiv beregning af 2. ordens information fra fejlfunktionen. Det har været den gængse opfattelse, at 2. ordens information er for tidskrævende at benytte, fordi det umiddelbart ser ud til at kræve beregning af Hessian matricen. Det viser sig nu, at der er mange muligheder for at trække information ud uden explicit at skulle beregne hele matricen. I dette kapitel præsenteres resultaterne fra appendix C og D.

1.1.4 Kapitel 5

Kapitel 5 beskriver problemstillinger omkring brug af fejlfunktion (objektfunktion) til træning af neurale netværk. Det bliver understreget, at de gængse funktioner, så som least mean square og entropy funktionen, ofte ikke er hensigtsmæssige til neurale netværks træning. I den forbindelse præsenteres resultaterne fra appendix E, hvor en alternativ fejlfunktion beskrives.

1.1.5 Kapitel 6

Kapitel 6 giver en overordnet konklusion af arbejdet. Her konkluderes bla., at *algoritmen* til træning af neurale netværk stadig ikke eksisterer, men at det med afhandlingen er blevet nemmere at vælge en passende algoritme alt efter problemstillingen.

Man skal overveje, om træning skal foregå i *on-line* eller *off-line* mode. On-line er som regel bedst på klassifikationsproblemer med store, redundante træningssæt. Hvis on-line mode bliver valgt, står valget mellem stochastic scaled conjugate gradient algoritmen (se sektion 3.2.4), eller en omhyggelig "tunet" on-line gradient descent algoritme kombineret med teknikker som beskrevet i sektion 3.1.7. Hvis off-line mode bliver valgt, bør valget falde på en 2. ordens metode, som scaled conjugate gradient algoritmen beskrevet i sektion 3.2.3.

Der konkluderes også, at 2. ordens information *kan* beregnes uden explicit at skulle beregne hele Hessian matricen. Sådanne teknikker er yderst lovende og kan have stor betydning for udvikling af bedre træningsmetoder samt betydning for metoder til optimering af netværksarkitektur.

1.2 Artikel 1

Artiklen "A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning" er blevet udgivet i tidsskriften *Neural Networks* (Vol. 6, pp. 525-533, 1993).

Artiklen giver en introduktion til konjugerede gradient algoritmer. Konjugerede gradient algoritmer er yderst velegnede til store optimerings problemer, idet de involverer 2. ordens information uden explicit at involvere Hessian matricen. Problemet med brugen af konjugerede gradient algoritmer til træning af neurale netværk har været, at disse involverer en linie søgning til estimering af en passende skridtlængde. En sådan liniesøgning er tidskrævende, idet den involverer flere beregninger af fejlen og/eller gradienten til fejlen.

Artiklen præsenterer en ny variation af en konjugeret gradient algoritme, der undgår denne liniesøgning. Algoritmen introducerer en skalerings mekanisme i stil med den fundet i Levenberg-Marquardt algoritmen og kombinerer denne med en "model-trust region" tilgang. Algoritmen udkonkurrerer andre eksisterende konjugerede gradient algoritmer samt gradient descent på flere test problemer.

1.3 Artikel 2

Artiklen "Supervised Learning on Large Redundant Training Sets" er blevet udgivet i tidsskriftet *International Journal of Neural Systems* (Vol. 4, pp. 15-25, 1993).

SCG algoritmen, som beskrevet i artikel 1, er en off-line algoritme, hvilket betyder, at alle data skal processeres gennem netværket før en opdatering kan foregå. På store, redundante datasæt kan dette alvorligt hæmme konvergensen set i forhold til algoritmer, der kan opdatere on-line efter processering af enkelte datamønstre. Artiklen præsenterer en stokastisk version af SCG algoritmen, som kan opdatere på mindre blokke af data.

1.4 Artikel 3

Artiklen "Exact Calculation of the Product of the Hessian Matrix and a Vector in O(N) Time" er blevet indsendt til tidsskriftet *Neural Computation* og udgivet som en teknisk rapport på DAIMI, Aarhus Universitet.

I flere sammenhænge er det nødvendigt at kende Hessian matricen gange en vektor. Dette gælder f.eks. i SCG algoritmen samt i estimering af egenværdier til Hessian matricen. Hessian gange en vektor kan approximeres numerisk ved en en-sidet difference ligning, men indtil fornylig har det ikke været mulig at beregne denne størrelse eksakt uden først eksplicit at skulle beregne Hessian matricen.

Artiklen giver en algoritme til eksakt beregning af Hessian matricen gange en vektor, og beviser at denne er korrekt. Algoritmen opererer i samme tidsorden som beregning af gradienten til fejlfunktionen. En tilsvarende algoritme er uafhængigt og på samme tid blevet udviklet af Barak Pearlmutter, Siemens Corporation Research, Princeton.

1.5 Artikel 4

Artiklen "Adaptive Preconditioning of the Hessian Matrix" er blevet indsendt til tidsskriftet *Neural Computation*.

Konditionstallet af Hessian matricen har en afgørende indvirkning på konvergensen af gradient descent samt konjugerede gradient algoritmer. En velkendt teknik til at forbedre konvergensen i konjugerede gradient algoritmer er prækonditionering af Hessian matricen, hvor denne transformeres vha. af en passende *prækonditioneringsmatrice*. De gængse metoder virker kun på positive definite matricer og er for tidskrævende i neurale netværks sammenhæng.

Artiklen beskriver problemerne omkring prækonditionering af indefinite matricer og præsenterer en ny metode til adaptiv prækonditionering af Hessian matricen under træning. Metoden illustreres ved eksempler.

1.6 Artikel 5

Artiklen "Supervised Learning: Improving Neural Network Solutions" er blevet lavet i samarbejde med Scott Fahlman, Carnegie Mellon University, Pittsburgh. En modificeret udgave af artiklen er blevet indsendt til tidsskriftet *Neural Computation*.

Least mean square funktionen er en ofte brugt fejlfunktion til træning af neurale netværk. Ved klassificeringsproblemer er denne langt fra optimal, idet minimering af fejlen ikke nødvendigvis medfører minimering af fejlklassificeringer. Funktionen er ikke *monoton* mht. klassifikation.

Artiklen præsenterer en ny fejlfunktion, som er *soft-monoton*, dvs. "graden" af monotoni kontrolleres af en bruger afhængig parameter. I den forbindelse præsenteres også en benchmark test mellen SCG algoritmen og Quickprop algoritmen.

Chapter 2

Notation and basic definitions

The networks we consider are multilayered feed-forward neural networks with arbitrary connectivity. The network $\aleph$ consist of nodes n_m^l arranged in layers $l = 0, \ldots, L$. The number of nodes in a layer l is denoted N_l. In order to be able to handle the arbitrary connectivity we define for each node n_m^l a set of *source nodes* and a set of *target nodes*.

$$\begin{aligned} S_m^l &= \left\{ n_s^r \in \aleph \mid n_s^r \text{ connects to } n_m^l, \;\; r < l, \;\; 1 \leq s \leq N_r \right\} \\ T_m^l &= \left\{ n_s^r \in \aleph \mid n_m^l \text{ connects to } n_s^r, \;\; r > l, \;\; 1 \leq s \leq N_r \right\} \end{aligned} \tag{2.1}$$

The training set associated with network $\aleph$ is

$$\left\{ (u_{ps}^0, s = 1, \ldots, N_0, \;\; t_{pj}, j = 1, \ldots, N_L), p = 1, \ldots, P \right\} \tag{2.2}$$

The output from a node n_m^l when a pattern p is propagated through the network is

$$u_{pm}^l = f(v_{pm}^l) \text{ , where } v_{pm}^l = \sum_{n_s^r \in S_m^l} w_{ms}^{lr} u_{ps}^r + w_m^l, \tag{2.3}$$

and w_{ms}^{lr} is the weight from node n_s^r to node n_m^l. w_m^l is the usual *bias* of node n_m^l. $f(v_{pm}^l)$ is an appropriate activation function, e.g., hyperbolic tangent. The net-input v_{pm}^l is chosen to be the usual weighted linear summation of inputs. The calculations to be made could, however, easily be extended to other definitions of v_{pm}^l. Let an error function $E(\mathbf{w})$ be

$$E(\mathbf{w}) = \sum_{p=1}^{P} E_p(u_{p1}^L, \ldots, u_{pN_L}^L, t_{p1}, \ldots, t_{pN_L}) \text{ ,} \tag{2.4}$$

where $\mathbf{w}$ is a vector containing all weights and biases in the network, and E_p is some appropriate error measure associated with pattern p from the training set. Coordinates of vectors and matrices will depending on the context also be referred to by the simpler notation $[\mathbf{w}]_i$ and $[A]_{ij}$.

The gradient vector $E^{'}(\mathbf{w})$ of an error function $E(\mathbf{w})$ is an $N \times 1$ vector given by

$$E^{'}(\mathbf{w}) = \sum_{p=1}^{P} \left(\frac{\partial E_p}{\partial [\mathbf{w}]_1}, \ldots, \frac{\partial E_p}{\partial [\mathbf{w}]_N} \right)^T . \tag{2.5}$$

The Hessian matrix $E^{''}(\mathbf{w})$ of an error function $E(\mathbf{w})$ is

$$E^{''}(\mathbf{w}) = \sum_{p=1}^{P} \begin{pmatrix} \frac{\partial^2 E_p}{\partial [\mathbf{w}]_1^2} & \cdots & \frac{\partial^2 E_p}{\partial [\mathbf{w}]_1 \partial [\mathbf{w}]_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 E_p}{\partial [\mathbf{w}]_1 \partial [\mathbf{w}]_N} & \cdots & \frac{\partial^2 E_p}{\partial [\mathbf{w}]_N^2} \end{pmatrix} . \tag{2.6}$$

A set $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_N$ of vectors are said to be mutually conjugate with respect to a matrix A if

$$\mathbf{p}_i^T A \mathbf{p}_j = 0 \text{ , when } \quad i \neq j. \tag{2.7}$$

The condition number κ of a matrix A is

$$\kappa = \left| \frac{\lambda_{max}}{\lambda_{min}} \right| \ , \tag{2.8}$$

where λ_{max} and λ_{min} are the largest and smallest eigenvalue of A respectively.

Chapter 3

Training Methods for Feed-Forward Networks: An Overview

This chapter reviews different methods for training feed-forward neural networks. The viewpoint is that of optimization which allows us to use results from the optimization literature. These results give information about computational complexity, congergence rates and safety procedures to ensure convergence and avoid numerical instabilities. Throughout the chapter we use results from the well written paper about first- and second order methods written by Battiti [Battiti 92]. We emphasize that this review is not a full survey of all existing techniques to train feed-forward networks, but a presentation of material that puts the results presented in appendix A and B into a broader context.

The presentation will focus on methods which are especially well suited for training of feed-forward networks. Factors that are important in this classification are computational complexity and the number of problem dependent parameters. In the description of the different methods we separate between *first order* and *second order* methods, i.e., between methods based on a linear model and methods based on a quadratic model of the error function.

3.1 Gradient descent

Gradient descent is one of the oldest optimization methods known. The use of the method as a basis for multivariate function minimization dates back to Cauchy in 1847 [Cauchy 1847], and has been the subject of intense analysis. Gradient descent is based on a linear approximation of the error function given by

$$E(\mathbf{w} + \triangle\mathbf{w}) \approx E(\mathbf{w}) + \triangle\mathbf{w}^T E'(\mathbf{w}). \tag{3.1}$$

The weight update is

$$\triangle\mathbf{w} = \Leftrightarrow\eta E'(\mathbf{w})\ , \quad \eta > 0. \tag{3.2}$$

The step size or learning rate η can be determined by a line search method but is usually set to a small constant. In the latter case the algorithm is, however, not guaranteed to converge. If η is chosen optimally in each step the method is often called the *steepest descent method*. The method can be used in *off-line* or *on-line* mode. The off-line mode is the one presented in (3.2), where the gradient vector is an accummulation of partial gradient vectors, one for each pattern in the training set. In the on-line mode, gradient descent is performed successively on each partial error function associated with one given pattern in the training set. The update formula is then given by

$$\triangle \mathbf{w} = \Leftrightarrow \eta E_p^{'}(\mathbf{w}) \ , \quad \eta > 0, \tag{3.3}$$

where $E_p^{'}(\mathbf{w})$ is the error gradient associated with pattern p. If η tends to zero over time, the movement in weight space during one *epoch*[1] will be similar to the one obtained with one off-line update. However, in general the learning rate has to be large to accelerate convergence, so that the paths in weight space of the two methods differ. The on-line method is often preferable to the off-line method when the training set is large and contains redundant information. This is especially true on problems where the targets only have to be approximated such as classification problems. For further discussion about issues concerning on-line and off-line techniques see section 3.4.

3.1.1 Back-Propagation

Until only recently gradient descent was only applicable to single layer feed-forward networks, because a method for the calculation of the gradient $E^{'}(\mathbf{w})$ for multi-layer networks did not exist before that time. A method to calculate the gradient in general was derived independently several times, by Bryson and Ho [Bryson and Ho 69], Werbos [Werbos 74], Parker [Parker 85] and Rumelhart [Rumelhart et al. 86]. The method is now known as the *back-propagation method*, and is central to much current work on learning in neural networks. There is some confusion about what the name *back-propagation method* actually refers to in the literature. Some researchers connects the name to the calculation of the gradient $E^{'}(\mathbf{w})$, others use the name to refer to the gradient descent algorithm itself. We use the name to refer to the gradient calculation. The following lemma summarizes the back-propagation method.

Lemma 1 *The gradient $E_p^{'}(\mathbf{w})$ of one particular pattern p can be calculated by one forward and one backward propagation. The forward propagation formula is:*

$$u_{pm}^{l} = f(v_{pm}^{l}) \ , \ \textit{where}$$

$$v_{pm}^{l} = \textstyle\sum_{n_s^r \in S_m^l} w_{ms}^{lr} u_{ps}^{r} + w_m^l \ ,$$

The backward propagation formula is:

$$[E_p^{'}(\mathbf{w})]_{mi}^{lh} = \delta_{pm}^{l} u_{pi}^{h} \ , \qquad [E_p^{'}(\mathbf{w})]_m^l = \delta_{pm}^{l} \ ,$$

where δ_{pm}^{l} is given recursively by:

$$\delta_{pm}^{l} = f'(v_{pm}^{l}) \textstyle\sum_{n_s^r \in T_m^l} w_{sm}^{rl} \delta_{ps}^{r} \ \ , l < L \ , \qquad \delta_{pj}^{L} = \frac{\partial E_p}{\partial v_{pj}^{L}} \ \ , 1 \leq j \leq N_L .$$

The gradient of the error corresponding to the whole training set is of course a sum of the partial gradients calculated in lemma 1. Lemma 1 can easily be derived using the chain rule backwards in the network. The main idea of propagating error information back through the network can be extended to the calculation of other important error information features such as second order information. We will get back to that in chapter 4.

[1] An epoch is equal to one full presentation of all patterns in the training set.

Figure 3.1: The steepest descent trajectory on a simple quadratic surface. Notice that the search directions are perpendicular to each other and to the tangent planes of the contours.

3.1.2 Convergence rate

We now turn to the rate of convergence of the gradient descent algorithm. Considering that the negative gradient is the direction of fastest decrease in error, we would intuitively expect to get a fast convergence, if the learning rate η is chosen optimally. This is, however, not the case. If we assume that the error is locally quadratic, then the contours of $E(\mathbf{w})$, given by $E(\mathbf{w}) = c$ are N dimensional ellipsoids with the minimum of the quadratic as the center. The axes of the ellipsoids are in the direction of the N-mutually orthogonal eigenvectors of the Hessian and the length of the axes are equal to the inverse of the corresponding eigenvalues [Luenberger 84]. The gradient $E^{'}(\mathbf{w})$ in a point $\mathbf{w}$ on the ellipsoid is perpendicular to the tangent plane in $\mathbf{w}$. This means that the gradient descent direction $\Leftrightarrow E^{'}(\mathbf{w})$ will not in general point directly to the minimum of the quadratic (the center of the ellipsoid). The search directions chosen tend to interfere so that a minimization in a current direction can ruin past minimizations in other directions. In fact, the minimization is done in a kind of "zig-zagging" way, where the current search direction is perpendicular to the last search direction. This is easily verified by the following. Let $\mathbf{d}_k = \Leftrightarrow E^{'}(\mathbf{w}_k)$ be the direction of search in the kth iteration. If we minimize in the direction of $\mathbf{d}_k$ with respect to η, then

$$\frac{d}{d\eta}E(\mathbf{w}_k + \eta\mathbf{d}_k) = \mathbf{d}_k^T E^{'}(\mathbf{w}_k + \eta\mathbf{d}_k) = 0 \qquad \Rightarrow \qquad \mathbf{d}_k^T\mathbf{d}_{k+1} = 0$$

Figure 3.1 illustrates the whole situation. As we shall see in section 3.2, one of the major advantages with conjugate gradient methods is that they approximate non-interfering directions of search.

If we assume that the error function is quadratic with constant and positive definite Hessian, [2] then it is possible to show that the condition number of the Hessian matrix has a major impact on the convergence rate. We can write the error in a neighborhood of a point $\mathbf{w}$ as

$$E(\mathbf{w}_k) = E(\mathbf{w}) + (\mathbf{w}_k \Leftrightarrow \mathbf{w})^T E^{'}(\mathbf{w}) + \frac{1}{2}(\mathbf{w}_k \Leftrightarrow \mathbf{w})^T E^{''}(\mathbf{w})(\mathbf{w}_k \Leftrightarrow \mathbf{w}) \ . \qquad (3.4)$$

[2] This is, however, not always the case (see section 4).

If $\mathbf{w}_*$ is the minimum of the error, then $E'(\mathbf{w}) = \Leftrightarrow E''(\mathbf{w})(\mathbf{w}_* \Leftrightarrow \mathbf{w})$ and $E(\mathbf{w}_k)$ can be written in the alternative form

$$E(\mathbf{w}_k) = \frac{1}{2}(\mathbf{w}_k \Leftrightarrow \mathbf{w}_*)^T E''(\mathbf{w})(\mathbf{w}_k \Leftrightarrow \mathbf{w}_*) + E(\mathbf{w}) \Leftrightarrow \frac{1}{2}(\mathbf{w}_* \Leftrightarrow \mathbf{w})^T E''(\mathbf{w})(\mathbf{w}_* \Leftrightarrow \mathbf{w}) \ . \quad (3.5)$$

The last two terms of (3.5) are constants and can be ignored when minimizing $E(\mathbf{w}_k)$. If the learning rate η is chosen optimal then

$$\eta = \frac{E'(\mathbf{w}_k)^T E'(\mathbf{w}_k)}{E'(\mathbf{w}_k)^T E''(\mathbf{w}) E'(\mathbf{w}_k)}. \quad (3.6)$$

In order to obtain a bound of the convergence rate we need the following lemma.

Lemma 2 *The Kantorovitch-Bergstrom inequality states*

$$\frac{(\mathbf{x}^T A \mathbf{x})(\mathbf{x}^T A^{-1} \mathbf{x})}{(\mathbf{x}^T \mathbf{x})^2} \leq \frac{(\lambda_{max} + \lambda_{min})^2}{4 \lambda_{max} \lambda_{min}}$$

where A is a symmetric positive definite matrix and λ_{max} and λ_{min} are the largest and smallest eigenvalue respectively.

Proof. See [Aoki 71] or [Luenberger 84]. □
Based on lemma 2 we can now make the connection between convergence rate and condition number.

Lemma 3 *Assume that the error function is quadratic with constant Hessian $E''(\mathbf{w})$. At every step in the gradient descent algorithm it holds that*

$$\frac{E(\mathbf{w}_{k+1}) \Leftrightarrow E(\mathbf{w}_*)}{E(\mathbf{w}_k) \Leftrightarrow E(\mathbf{w}_*)} \leq \left(\frac{\kappa \Leftrightarrow 1}{\kappa + 1} \right)^2 ,$$

where κ is the condition number of $E''(\mathbf{w})$.

Proof. Using (3.6) and the fact that

$$\begin{aligned} E'(\mathbf{w}_k) &= E''(\mathbf{w})(\mathbf{w}_k \Leftrightarrow \mathbf{w}_*) \text{ and} \\ E(\mathbf{w}_k) \Leftrightarrow E(\mathbf{w}_*) &= \frac{1}{2} E'(\mathbf{w}_k)^T E''(\mathbf{w})^{-1} E'(\mathbf{w}_k) \ , \end{aligned}$$

we get

$$\begin{aligned} \frac{E(\mathbf{w}_k) \Leftrightarrow E(\mathbf{w}_{k+1})}{E(\mathbf{w}_k) \Leftrightarrow E(\mathbf{w}_*)} &= \frac{2\eta E'(\mathbf{w}_k)^T E''(\mathbf{w})(\mathbf{w}_k \Leftrightarrow \mathbf{w}_*) \Leftrightarrow \eta^2 E'(\mathbf{w}_k)^T E''(\mathbf{w}) E'(\mathbf{w}_k)}{E'(\mathbf{w}_k)^T E''(\mathbf{w})^{-1} E'(\mathbf{w}_k)} \\ &= \frac{(E'(\mathbf{w}_k)^T E'(\mathbf{w}_k))^2}{(E'(\mathbf{w}_k)^T E''(\mathbf{w}) E'(\mathbf{w}_k))(E'(\mathbf{w}_k)^T E''(\mathbf{w})^{-1} E'(\mathbf{w}_k))} \\ &\geq \frac{4 \lambda_{max} \lambda_{min}}{(\lambda_{max} + \lambda_{min})^2} \ , \end{aligned}$$

where λ_{max} and λ_{min} are the largest and smallest eigenvalue of $E''(\mathbf{w})$. The result is now easily derived from the last inequality [Luenberger 84]. □

Lemma 3 states that the gradient descent method converges linearly with a ratio not greater than $\left(\frac{\kappa-1}{\kappa+1}\right)^2$. It can be shown that if the condition number κ is high, then the method is very likely to converge at a rate close to the bound [Akaike 59]. So the bigger difference between the largest and smallest eigenvalue, the slower convergence of the gradient descent method. Geometrically this means that the more the contours of $E(\mathbf{w}_k)$ are skewed the slower the convergence of gradient descent. Even if only one eigenvalue is large and all others are of equal size, the convergence will be slow.

3.1.3 Gradient descent with momentum

In [Plaut et al. 86] the gradient descent update is changed to

$$\triangle\mathbf{w}_{k+1} = \Leftrightarrow\eta E'(\mathbf{w}_k) + \alpha\triangle\mathbf{w}_k \ , \quad \eta > 0 \ , \ 0 < \alpha < 1 \ , \tag{3.7}$$

where α is the so called *momentum* term. The addition of this momentum term incorporates second order information into the method since current as well as past gradient information is taken into account. The change was introduced to avoid oscillations in narrow steep regions of weight space and to increase convergence in flat regions. In [Watrous 87] an analysis of the effect of the momentum term was given. We shortly summarize these results. The weight update formula given by (3.7) is a special version of a first order difference equation [Press et al. 88],[3] which solution is given by

$$\triangle\mathbf{w}_{k+1} = \alpha^k\triangle\mathbf{w}_1 \Leftrightarrow\eta\sum_{i=1}^{k}\alpha^{k-i}E'(\mathbf{w}_i) \ . \tag{3.8}$$

Written in terms of the weights this becomes

$$\mathbf{w}_{k+1} = \mathbf{w}_k \Leftrightarrow\eta\sum_{i=0}^{k}\alpha^{k-i}E'(\mathbf{w}_i) \tag{3.9}$$

Equation (3.9) is a first order difference equation in $\mathbf{w}$ with solution

$$\mathbf{w}_{k+1} = \mathbf{w}_0 \Leftrightarrow\eta\sum_{j=0}^{k}\sum_{i=0}^{k}\alpha^{j-i}E'(\mathbf{w}_i) = \mathbf{w}_0 \Leftrightarrow\eta\sum_{j=0}^{k}\alpha^j\sum_{i=0}^{k-j}E'(\mathbf{w}_i) \tag{3.10}$$

In flat regions in weight space the gradient can be approximated with a constant, say E'. Under this assumption we have

$$\mathbf{w}_{k+1} = \mathbf{w}_0 \Leftrightarrow\eta E'\sum_{j=0}^{k}(k\Leftrightarrow j+1)\alpha^j \tag{3.11}$$

Splitting equation (3.11) up in two finite sums,[4] and evaluating we get

$$\begin{aligned} \mathbf{w}_{k+1} &= \mathbf{w}_0 \Leftrightarrow\eta E'\left((k+1)\frac{1\Leftrightarrow\alpha^{k+1}}{1\Leftrightarrow\alpha} \Leftrightarrow\frac{\alpha(1\Leftrightarrow\alpha^k)}{(1\Leftrightarrow\alpha)^2} + \frac{k\alpha^{k+1}}{1\Leftrightarrow\alpha}\right) \\ &= \mathbf{w}_0 \Leftrightarrow\eta E'\left(\frac{k+1}{1\Leftrightarrow\alpha}\right)\left(1\Leftrightarrow\frac{1\Leftrightarrow\alpha^{k+1}}{k+1}\frac{\alpha}{1\Leftrightarrow\alpha}\right) . \end{aligned} \tag{3.12}$$

[3] The solution for the general equation $x_{k+1} = a_k x_k + b_k$ is: $x_{k+1} = \prod_{j=1}^{k} a_j x_1 + \sum_{i=1}^{k}\prod_{j=i+1}^{k} a_j b_i$.

[4] We here use that: $\sum_{j=0}^{k} x^j = \frac{1-x^{k+1}}{1-x}$ and $\sum_{j=0}^{k} jx^j = \frac{x(1-x^k)}{(1-x)^2} - \frac{kx^{k+1}}{1-x}$, $|x| < 1$.

The effect of α is now clear. In flat regions of weight space the convergence rate is accelerated with a factor approaching $\frac{1}{1-\alpha}$, when k gets large. In narrow steep regions the effect of α is to average out components of the gradient with alternating signs.

As we shall see in section 3.2, gradient descent with momentum is an approximation of a conjugate gradient update. In conjugate gradient methods, however, η and α are chosen automatically. The problem with the gradient descent with momentum is that η and α has to be guessed by the user. Furthermore, the optimal values of η and α might change in each iteration.

3.1.4 Adaptive learning rate and momentum

Many heuristic schemes to adapt the learning rate and/or the momentum dynamically have been proposed in the literature, such as [Cater 87], [Franzini 87], [Chan and Fallside 87], [Jacobs 88], [Vogl et al. 88], [Battiti 89], [Silva and Almeida 90] and [Tollenaere 90]. It will go too far to describe them all here. We will, however, briefly describe some of the best known approaches.

One heuristic to adapt both learning rate and momentum has been proposed by Chan and Fallside [Chan and Fallside 87]. The main idea of the learning rate adaptation is to calculate the angle θ_k between the current gradient and the last weight update and use this as information about the error surface characteristics. If $90° \leq \theta_k \leq 270°$, arrival at a ravine wall is likely and the learning rate should be decreased. If θ_k approaches 0° or 360°, arrival at a plateau is likely and the learning rate should be increased. Chan and Fallside suggest the following adaptation of η.

$$\eta_k = \eta_{k-1}(1 + \frac{1}{2}\cos\theta_k) \tag{3.13}$$

When a constant momentum is used, the weight update vector can be dominated by the momemtum term and even point uphill instead of downhill. The idea of the adaptation of the momentum is therefore to insist of having the magnitude of the momentum term smaller than the magnitude of the gradient term. In this case the gradient term will always be the dominating factor in the weight update vector. The adaptation of the momentum is given by

$$\alpha_k = \alpha_0 \eta_k \frac{|\triangle E(\mathbf{w}_k)|}{|\triangle \mathbf{w}_{k-1}|}\ ,\quad 0 < \alpha_0 < 1. \tag{3.14}$$

This method yields good results compared to standard gradient descent, but is however not without problems. The setting of η_0 and α_0 might be crucial for the success of this adaptation scheme. Chan and Fallside incorporates a backtracking scheme into the algorithm to prevent too large learning rates caused from too high an initial η_0 value. If the error is larger than the previous error, the learning rate η_k is then reduced by a half. See [Chan 90] for a comparison of this method with other adaptive methods.

Several researchers have explored the idea of having a learning rate and/or a momentum for each unit or even for each weight in the network. The motivation for this strategy is that parameters appropriate for any one weight dimension might not be appropriate for other dimensions. Note that having different learning rates for each unit or each weight, means that the weights are not modified in the direction of the negative gradient any longer. Thus, such a system is not doing gradient descent any more. Instead, the weights are updated based on gradient information together with estimated information about the curvature.

One scheme, that has learning rates and momentum for each unit is described in [Haffner et al. 88]. They develop heuristic schemes for adaptation of both learning rate and momentum. The idea for the learning rate scheme is to limit the norm of the learning rate times the gradient to a fixed value, say ω. This can be achieved by adapting the learning rate η_m^l associated with unit number m in layer l with

$$\eta_m^l = \frac{\eta}{1 + \frac{\eta}{\omega}\sqrt{\sum_{n_s^r \in S_m^l} \left(\frac{dE}{dw_{ms}^{lr}}\right)^2}}, \quad \eta > 0, \;\; \omega > 0. \tag{3.15}$$

Haffner *et al.* states that a value of ω equal to one yields good results. They do, however, not say anything about the value of the other user-dependent parameter η, which value might be crucial for the convergence rate.

The momentum term is adapted in a similar fashion. The overall idea is, that the more the network changes the smaller should the momentum be. A characteristic symptom of too large a momentum is divergence of the term $|\mathbf{w}|^2$ over time. Thus in the kth iteration and for each unit, Haffner *et al.* defines a control measure by

$$\begin{aligned} Q_m^l(k) &= \sum_{n_s^r \in S_m^l} [w_{ms}^{lr}(k)]^2 \Leftrightarrow \sum_{n_s^r \in S_m^l} [w_{ms}^{lr}(k \Leftrightarrow 1)]^2 \\ &= 2 \sum_{n_s^r \in S_m^l} w_{ms}^{lr}(k \Leftrightarrow 1) \triangle w_{ms}^{lr}(k) + \sum_{n_s^r \in S_m^l} [\triangle w_{ms}^{lr}(k)]^2 \end{aligned} \tag{3.16}$$

In order to limit $Q_m^l(k)$, the momentum is chosen such as to be inverse proportional to the first term in (3.16). The adaptation formula is

$$\alpha_m^l = \frac{1}{1 + \psi |\sum_{n_s^r \in S_m^l} w_{ms}^{lr}(k \Leftrightarrow 1) \triangle w_{ms}^{lr}(k)|}, \quad \psi > 0. \tag{3.17}$$

This adaptation formula does not ensure that the control measure $Q_m^l(k)$ is limited to a fixed value as was the case for learning rate adaptation. Through several experiments Haffner *et al.* concludes that the method yields faster convergence than standard gradient descent and also gives a higher percentage of converging trials.

Another heuristic method of adapting learning rates is the *delta-bar-delta* method proposed by Jacobs [Jacobs 88]. In this case there are independent learning rates for each single weight. Jacobs develops a gradient descent like updating rule for the learning rates for each unit in the network. Let Θ be a diagonal matrix of learning rate values. Then we can approximate the derivative $\frac{dE}{d\Theta}$ by

$$\frac{dE}{d\Theta} \approx \frac{dE}{d\mathbf{w}_k^T} \frac{d\mathbf{w}_k}{d\Theta} \tag{3.18}$$

The derivative with respect to each learning rate η_{ms}^{lr} is then given by

$$\frac{\delta E}{\delta \eta_{ms}^{lr}} = \Leftrightarrow \frac{\delta E}{\delta w_{ms}^{lr}(k)} \frac{\delta E}{\delta w_{ms}^{lr}(k \Leftrightarrow 1)}. \tag{3.19}$$

$\frac{dE}{d\Theta}$ can then be updated simultanously with the weights by the gradient descent update rule

$$\triangle\eta^{lr}_{ms} = \gamma \frac{\delta E}{\delta w^{lr}_{ms}(k)} \frac{\delta E}{\delta w^{lr}_{ms}(k \Leftrightarrow 1)}\ ,\quad \gamma > 0. \tag{3.20}$$

This update scheme is called the *delta-delta* rule and is, unfortunately, of limited practical use. The problem is, that the convergence of the process is crucially dependent of the value of γ. Jacobs overcomes this problem, by defining a new update rule, which only in principle works in a similar fashion as the delta-delta rule. The delta-bar-delta rule is given by

$$\triangle\eta^{lr}_{ms} = \begin{cases} \gamma & \overline{\delta}_{k-1}\delta_k > 0\ ,\quad \gamma > 0 \\ \Leftrightarrow\phi\eta^{lr}_{ms} & \overline{\delta}_{k-1}\delta_k < 0\ ,\quad \phi > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.21}$$

where $\delta_k = \frac{\delta E}{\delta w^{lr}_{ms}(k)}$ and $\overline{\delta}_k = (1 \Leftrightarrow \theta)\delta_k + \theta\overline{\delta}_{k-1}$,$0 < \theta < 1$, i.e. $\overline{\delta}_k$ is a running average of the current and past gradients. If the current gradient has opposite sign as the running average gradient the learning rate is decreased exponentially. If the current gradient has the same sign as the running average gradient then the learning is increased linearly. The difference between the delta-delta rule and the delta-bar-delta rule is that the latter takes average gradients into account and updates the learning rates independently of the size of the current gradient.

Jacobs reports a significant increase of convergence compared to standard gradient descent. There is, however, some problems with the delta-bar-delta method that are unclarified. A problem that immediately can be identified, is how to select the values of the two user-dependent parameters γ and ϕ. The value of these parameters might be very crucial for the success of this scheme. Jacobs does not give any description of how to set these parameters.

3.1.5 Learning rate schedules for on-line gradient descent

In this section we present some promising learning rate schedules introduced by Darken *et al.* [Darken et al. 92]. These schedules are only functions of time and not of previous values of learning rates or other parameters. The schedules are based on results from stochastic approximation theory, see for example [Robbins and Monro 51] and [Goldstein 87].

In standard stochastic approximation theory results are given about the convergence properties of on-line gradient descent. On-line gradient descent on the least mean square error function is guaranteed to converge if the learning rate η_k satisfies

$$\sum_{k=1}^{\infty} \eta_k = \infty\ ,\quad \text{and} \quad \sum_{k=1}^{\infty} \eta_k^2 = 0. \tag{3.22}$$

When the learning rate η_k is only a function of time, it can be shown that the optimal rate of convergence is proportional to k^{-1}, i.e., $|\mathbf{w}_k \Leftrightarrow \mathbf{w}_*|^2 \propto k^{-1}$, where $\mathbf{w}_*$ is the desired minimum. When the learning rate is allowed to depend on current or previous values of the learning rate or other parameters, as in the adaptive schemes described in the last section, very little is known theoretically about the optimal convergence rate. In [Goldstein 87] it is shown that in order to converge at an optimal rate, we must have $\eta_k \rightarrow \frac{c}{k}$ asymptotically, for c greater than some threshold c_*, which depends on the error function and on the training set. Chung shows that c_* is equal to $\frac{1}{2\lambda_{min}}$, where λ_{min} is the smallest eigenvalue of the Hessian of the error function [Chung 54]. The usual choice of learning rate schedule in stochastic approximation theory is $\eta_k = \frac{c}{k}$. However, this scheme often converges slowly. Darken *et al.* propose a more sophisticated schedule that guarantees asymptotically optimal rate of convergence. The schedule is called *Search-Then-Converge (STC)* and is given by

$$\eta_k = \eta_0 \frac{1 + \frac{c}{\eta_0}\frac{k}{\tau}}{1 + \frac{c}{\eta_0}\frac{k}{\tau} + \tau\frac{k^2}{\tau^2}} \tag{3.23}$$

The main idea of this schedule is to delay the major decrease in the learning rate until a minimum has been located. η_k is approximately equal to η_0 at times small compared to $\sqrt{\tau}$, this is called the "search phase". For times greater than $\sqrt{\tau}$, the learning rate decreases as $\frac{c}{k}$, which is called the "convergence phase". Darken *et al.* demonstrates major improvements in convergence rate using this schedule compared to traditional learning rate schedules. There are, however, some problems with the method of setting initial parameters such as c, η_0 and τ. Darken *et al.* addresses the problem of setting the c parameter. As mentioned above c should be greater than $c_* = \frac{1}{2\lambda_{min}}$. In fact, Darken *et al.* shows that the system exhibits a kind of *phase transition* at $c = c_*$. This means that an arbitrarily small change in c, which moves it to the opposite side of c_* has a dramatic effect on the behaviour of the system. Darken *et al.* argue that using a direct method of estimating c_* is too time consuming since this involves estimation of the smallest eigenvalue of the Hessian. For this reason, they outline an *ad hoc* method of determining whether a particular value of c is less than c_*. They use a heuristic scheme to adapt c based on characteristics of the weight vector trajectory. It is, however, in our opinion an open question whether the direct method of estimating the smallest eigenvalue is too time consuming afterall. The smallest eigenvalue of the Hessian can be estimated by use of the Power method on the matrix $B = (\beta I \Leftrightarrow E''(\mathbf{w}_k))$, where $\beta > \lambda_{max}$ [Ralston et al. 78] (see also chapter 4 and appendix C). This estimation can be relaxed to an on-line situation by replacing B with a running average of the form

$$B_k = (1 \Leftrightarrow \gamma)B_{k-1} + \gamma(\beta I \Leftrightarrow E_p''(\mathbf{w}_k)) \ , \ \ 0 < \gamma < 1. \tag{3.24}$$

If the Power method is run simultanously with the update of weights, then this scheme would cost $O(N)$ time more per weight update, i.e. twice as much calculation work per iteration [Pearlmutter 93], [Møller 93c].

Darken *et al.* do not address the setting of the parameters η_0 and τ, because these values do not affect the asymptotic behavior of the system. The values do, however, have a major affect on the rate of convergence and methods for the initial setting of these parameters are needed in order for the method to have any real practical use. Some practical hints about how to set these parameters are the following [Darken 93]. η_0 should be taken as large as possible, but avoiding instability. $\sqrt{\tau}$ should be some fraction of the total number of iterations that are planned to be run. In the setting of τ, prior knowledge about the problem can be used. This could be knowledge about how noisy the problem is, or if it is easy to locate a minimum and so forth.

3.1.6 The quickprop method

Quickprop is not strictly a gradient descent method, but can be viewed somewhere between a gradient descent method and a Newton method [Fahlman 89]. It has, in our opinion not received the attention it deserves. We give a detailed description here.

In order to avoid the time and space consuming calculations involved with the Newton algorithm two approximations are made. The Hessian matrix is approximated by ignoring all non-diagonal terms making the assumption that all the weights are independent. Each term in the diagonal is approximated by a one sided difference formula given by

$$\frac{d^2E(w)}{dw^2} \approx \frac{E^{'}(w_k) \Leftrightarrow E^{'}(w_{k-1})}{w_k \Leftrightarrow w_{k-1}} \tag{3.25}$$

where w_k is a given weight at time step k. $\frac{d^2E(w)}{dw^2}$ can actually be calculated precisely with a little more calculation work [Le Cun 89], but is in the quickprop method not more efficient than the approximation. Geometrically the two approximations can be interpreted as follows. The error versus weight curve for each weight is approximated by a quadratic, which is assumed to be independent on changes in other weights. The main idea is to compute the minimum of this quadratic for each weight and update the weights by the following formula

$$\triangle w_k = \Leftrightarrow(\eta_k E^{'}(w_k) + \alpha_k)\ , \tag{3.26}$$

where η_k is

$$\eta_k = \begin{cases} \eta_0 & E^{'}(w_k)E^{'}(w_{k-1}) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.27}$$

and α_k is

$$\alpha_k = \begin{cases} \frac{w_k - w_{k-1}}{E^{'}(w_k) - E^{'}(w_{k-1})} E^{'}(w_k) & \left| \frac{E^{'}(w_k) - E^{'}(w_{k-1})}{E^{'}(w_{k-1})} \right| < \frac{\mu}{1+\mu} \\ \mu \triangle w_{k-1} & \text{otherwise} \end{cases} \tag{3.28}$$

The constant η_0 is similar to the learning rate in gradient descent. If $E^{'}(w_k)E^{'}(w_{k-1}) > 0$, i.e., the minimum of the quadratic has not been passed, a linear term is added to the quadratic weight change. On the other hand, if $E^{'}(w_k)E^{'}(w_{k-1}) \leq 0$, i.e., the minimum of the quadratic has been passed, only the quadratic weight change is used to go straight down to the minimum. μ is usually set equal to 2, which seems to work well in most applications.

The algorithm is usually used combined with a *primeoffset* term added to the first derivative of the sigmoid activation function. As noted in chapter 4 and appendix E, the use of primeoffset can influence the quality of the solutions found. Despite the two very crude approximations the quickprop algorithm has shown very good performance in practice. One drawback with the algorithm is, however, that the η_0 parameter is very problem dependent. An adaptive scheme to estimate this parameter would significantly increase the usefullness of this method. It might be possible to combine one of the adaptive learning rate schemes, described above to adapt η_0.

3.1.7 Estimation of optimal learning rate and reduction of large curvature components

The eigenvalues of the Hessian matrix can be interpreted as the curvature in the direction of the corresponding eigenvectors. The eigenvectors are the main axes of the contours of equal error, which are approximately ellipsoids with the minimum of the error as a center. Only if all the eigenvalues are of equal size, does the gradient descent direction point directly to the minimum (see figure 3.1). So as concluded in the analysis of the convergence rate of the gradient descent method, the main factor limiting the convergence rate of gradient descent is that the curvature of the error has different values in different directions. The largest curvature limits the maximum value of the learning rate, while the smallest curvature dominates the learning time. The optimal learning rate for off-line gradient descent can be shown to be equal to the inverse of the largest eigenvalue. In this section we describe methods to reduce the influence of large curvature components and also how to estimate the optimal learning rate.

The true direction to the minimum can be computed by multiplying the gradient descent vector by the inverse of the Hessian matrix, assuming that the Hessian is invertible. We then get the Newton direction (see chapter 3.3). The inverse of the Hessian can be expressed in terms of the eigenvectors and corresponding eigenvalues. By the spectral theorem from linear algebra we have that $E''(\mathbf{w}_k)$ has N eigenvectors that form an orthogonal basis in $\Re^N$ [Horn and Johnson 85]. This implies that the inverse of the Hessian matrix $E''(\mathbf{w}_k)^{-1}$ can be written in the form

$$E''(\mathbf{w}_k)^{-1} = \sum_{i=1}^{N} \frac{\mathbf{e}_i \mathbf{e}_i^T}{|\mathbf{e}_i|^2 \lambda_i} \ , \tag{3.29}$$

where λ_i is the ith eigenvalue of $E''(\mathbf{w}_k)$ and $\mathbf{e}_i$ is the corresponding eigenvector. Equation (3.29) implies that the Newton search directions $\mathbf{d}_k$ can be written as

$$\mathbf{d}_k = \Leftrightarrow E''(\mathbf{w}_k)^{-1} E'(\mathbf{w}_k) = \Leftrightarrow \sum_{i=1}^{N} \frac{\mathbf{e}_i \mathbf{e}_i^T}{|\mathbf{e}_i|^2 \lambda_i} E'(\mathbf{w}_k) = \Leftrightarrow \sum_{i=1}^{N} \frac{\mathbf{e}_i^T E'(\mathbf{w}_k)}{|\mathbf{e}_i|^2 \lambda_i} \mathbf{e}_i \ , \tag{3.30}$$

where $E'(\mathbf{w}_k)$ is the gradient vector. So the Newton search direction can be interpreted as a sum of projections of the gradient vector onto the eigenvectors weighted with the inverse of the eigenvalues. To calculate all eigenvalues and corresponding eigenvectors costs $O(N^3)$ time which is infeasible for large N. Le Cun *et al.* argues that only a few of the largest eigenvalues and the corresponding eigenvectors are needed to achieve a considerable speed up in learning. The idea is to reduce the weight change in directions with large curvature, while keeping it large in all other directions. A choice of search direction could be

$$\mathbf{d}_k = \Leftrightarrow \Big(E'(\mathbf{w}_k) \Leftrightarrow \mu \sum_{i=1}^{k} \frac{\mathbf{e}_i^T E'(\mathbf{w}_k)}{|\mathbf{e}_i|^2} \mathbf{e}_i \Big) \ , \quad 0 \le \mu \le 1. \tag{3.31}$$

where i now runs from the largest eigenvalue λ_1 down to the kth largest eigenvalue λ_k, and μ is some appropriate constant (Le Cun *et al.* suggest $\mu = \frac{\lambda_{k+1}}{\lambda_1}$). Equation (3.31) reduces the component of the gradient along the directions with large curvature. See also [Le Cun et al. 91] for a discussion of this. The learning rate can now be increased with a factor of $\frac{\lambda_1}{\lambda_{k+1}}$, since the components in directions with large curvature have been reduced with the inverse of this factor.

Another approach also proposed by Le Cun *et al.* is to use a small part of the sum in equation (3.30) as search direction, so that

$$\mathbf{d}_k = \Leftrightarrow \sum_{i=1}^{k} \frac{\mathbf{e}_i^T E'(\mathbf{w}_k)}{|\mathbf{e}_i|^2 \lambda_i} \mathbf{e}_i \ , \tag{3.32}$$

with $k \ll N$. In theory, this can accelerate the convergence by a factor $\frac{\lambda_1}{\lambda_{k+1}}$, compared to standard gradient descent.

The largest eigenvalue and the corresponding eigenvector can be estimated by an iterative process known as the *Power method* [Ralston et al. 78]. The Power method can be used successively to estimate the k largest eigenvalues if the components in the directions of already estimated eigenvectors are substracted in the process. Below we show an algorithm for estimation of the ith eigenvalue and eigenvector. The Power method is here combined with the *Rayleigh quotient technique* [Ralston et al. 78]. This can accelerate the process considerably.

Choose an initial random vector $\mathbf{e}_i^0$. Repeat the following steps for $m = 1, \ldots, M$, where M is a small constant:

$$\mathbf{e}_i^m = E''(\mathbf{w}_k)\mathbf{e}_i^{m-1} \;, \quad \mathbf{e}_i^m = \mathbf{e}_i^m \Leftrightarrow \sum_{j=1}^{i-1} \frac{\mathbf{e}_j^T \mathbf{e}_i^m}{|\mathbf{e}_j|^2} \mathbf{e}_j$$

$$\lambda_i^m = \frac{(\mathbf{e}_i^{m-1})^T \mathbf{e}_i^m}{|\mathbf{e}_i^{m-1}|^2} \;, \quad \mathbf{e}_i^m = \frac{1}{\lambda_i^m} \mathbf{e}_i^m \;.$$

λ_i^M and $\mathbf{e}_i^M$ are respectively the estimated eigenvalue and eigenvector. Theoretically it would be enough to substract the component in the direction of already estimated eigenvectors once, but in practice roundoff errors will generally introduce these components again. The term $E''(\mathbf{w}_k)\mathbf{e}_i^m$ can be approximated by a one-sided difference equation of the form

$$E''(\mathbf{w}_k)\mathbf{e}_i^m \approx \frac{E'(\mathbf{w}_k + \sigma \mathbf{e}_i^m) \Leftrightarrow E'(\mathbf{w}_k)}{\sigma} \quad ,0 < \sigma \ll 1 \tag{3.33}$$

See [Møller 93a] for an explanation. It has recently been shown independently by Pearlmutter and Møller that the term also can be calculated exactly in the same order of time as the approximation [Pearlmutter 93], [Møller 93c] (see also chapter 4).

The method of large curvature reduction can be relaxed to an on-line situation by replacing all terms of the form $E''(\mathbf{w})\mathbf{v}$, where $\mathbf{v}$ is a vector, with a running average of the form

$$E''(\mathbf{w})\mathbf{v} = (1 \Leftrightarrow \gamma)E''(\mathbf{w})\mathbf{v} + \gamma E_p''(\mathbf{w})\mathbf{v} \;, \;\; 0 < \gamma < 1, \tag{3.34}$$

where $E_p''(\mathbf{w})$ is the Hessian matrix associated with pattern p. Le Cun *et al.* reports significant increase in convergence even if only a few eigenvector are used.

Instead of changing the search direction $\mathbf{d}_k$, one can of course also use the above techniques to estimate the optimal learning rate $\eta = \lambda_{max}^{-1}$ for gradient descent. Le Cun *et al.* describes a technique based on equation (3.34) to esimate λ_{max}^{-1} and uses this as an estimate of the optimal on-line learning rate. It should here be noted, that the learning rate $\eta = \lambda_{max}^{-1}$ is not necessarily an optimal choice in the on-line version of gradient descent. Le Cun *et al.* reports, however, very impressive results with this scheme. The on-line estimation of the largest eigenvalue based on (3.34), seems to converge very quickly, i.e., after presentation of a small fraction of the whole training set. The largest eigenvalue seems to be mainly determined by network architecture and initial weigths, and by low-order statistics of the training data.

3.2 Conjugate Gradient

Conjugate gradient methods can be regarded as being somewhat intermediate between the method of gradient descent and Newton's method described in chapter 3.3. They are motivated by the desire to accelerate the typically slow convergence associated with the gradient descent method while avoiding the information requirements associated with the evaluation, storage and inversion of the Hessian matrix as required by the Newton method. The standard conjugate gradient method was originally developed by Hestenes and Stiefel to solve a set of equations with a positive definite matrix of coefficients [Hestenes and Stiefel 52]. Since then, conjugate gradient methods have become a standard method for non-linear function minimization.

In this section we give a brief introduction to the conjugate gradient methods and describe the convergence properties of the methods. The scaled conjugate gradient method is described in the last part of this section. We refer to appendix A-B for further details about conjugate gradient methods. In appendix A, an introductory part to conjugate gradient methods is also given. The introduction to follow differs deliberately from this introduction, in order to broaden the readers view of the methods and give additional insight.

3.2.1 Non-interfering directions of search

One of the problems with the gradient descent method was, that the gradient descent directions were interfering, so that a minimization in one direction could spoil past minimizations in other directions. This problem is solved in the conjugate gradient methods and is the heart of these methods. Under the assumption that the error function is quadratic, conjugate gradient methods produce non-interfering directions of search. This implies that in the kth iteration, the error has been minimized over the whole subspace spanned by all previous search directions. The necessary and sufficient condition to have non-interfering directions of search is, that the directions have to be mutually conjugate with respect to the Hessian matrix. So if $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_N$ is a set of directions, we have

$$\mathbf{p}_i^T E''(\mathbf{w})\mathbf{p}_j = 0 \text{ , when } \quad i \neq j. \tag{3.35}$$

This is easy to verify by the following. If we minimize the error optimally in the direction of say $\mathbf{p}_i$, then we have

$$\frac{d}{d\alpha}E(\mathbf{w}_i + \alpha\mathbf{p}_i) = 0 \quad \Rightarrow \quad E'(\mathbf{w}_{i+1})^T\mathbf{p}_i = 0 \tag{3.36}$$

In order to keep the error to be minimized in this direction, equation (3.36) has to be satisfied for all coming minimizations. So after minimization in a new direction $\mathbf{p}_j$, we need the condition

$$E'(\mathbf{w}_{j+1})^T\mathbf{p}_i = 0 \tag{3.37}$$

to be satisfied. Figure 3.2 illustrates the situation.

So we need to show that (3.35) ⇔ (3.37). This can be shown by induction. Assume that equation (3.37) is satisfied for all $\mathbf{w}_k$, $k < j+1$. The initial step, $k = i+1$, is immediately true by equation (3.36). Now $E(\mathbf{w}_{j+1})$ can be approximated by

$$E(\mathbf{w}_{j+1}) = E(\mathbf{w}_j + \alpha_j\mathbf{p}_j) \approx E(\mathbf{w}_j) + \alpha_j\mathbf{p}_j^T E'(\mathbf{w}_j) + \frac{1}{2}\alpha_j^2\mathbf{p}_j^T E''(\mathbf{w}_j)\mathbf{p}_j. \tag{3.38}$$

Observe that the α_j, that minimizes the quadratic (3.38) is given by

$$\alpha_j = \frac{\Leftrightarrow E'(\mathbf{w}_j)^T\mathbf{p}_j}{\mathbf{p}_j^T E''(\mathbf{w}_j)\mathbf{p}_j} \tag{3.39}$$

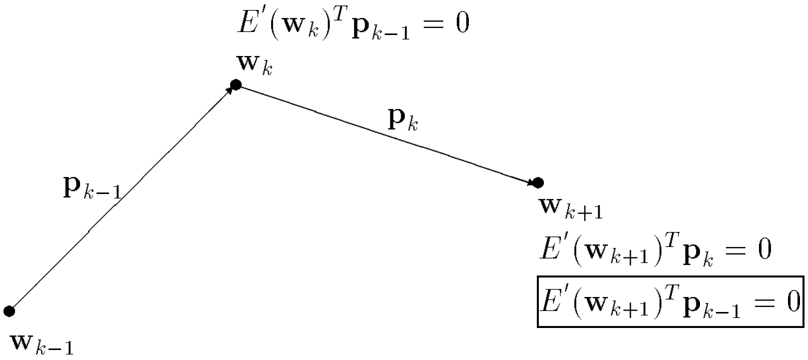
$E'(\mathbf{w}_k)^T\mathbf{p}_{k-1} = 0$
$\mathbf{w}_k$
$\mathbf{p}_k$
$\mathbf{p}_{k-1}$
$\mathbf{w}_{k+1}$
$E'(\mathbf{w}_{k+1})^T\mathbf{p}_k = 0$
$\boxed{E'(\mathbf{w}_{k+1})^T\mathbf{p}_{k-1} = 0}$
$\mathbf{w}_{k-1}$

Figure 3.2: Non-interfering direction $\mathbf{p}_{k-1}$ and $\mathbf{p}_k$. The formula in the box indicates the non-interference condition.

Differentiation of (3.38) gives

$$E'(\mathbf{w}_{j+1}) \approx E'(\mathbf{w}_j) + \alpha_j E''(\mathbf{w}_j)\mathbf{p}_j. \tag{3.40}$$

Multiplying by $\mathbf{p}_i$ gives the desired result

$$\begin{aligned} E'(\mathbf{w}_{j+1})^T\mathbf{p}_i &= 0 \quad \Leftrightarrow \\ \left(E'(\mathbf{w}_j) + \alpha_j \mathbf{p}_j^T E''(\mathbf{w}_j)\right)^T \mathbf{p}_i &= 0 \quad \Leftrightarrow \\ \mathbf{p}_j^T E''(\mathbf{w}_j)\mathbf{p}_i &= 0 \end{aligned} \tag{3.41}$$

The search directions can be determined recursively such that (3.35) is satisfied. The idea is to choose $\mathbf{p}_k$ to be the projection of the current gradient descent vector onto the subspace orthogonal to the subspace spanned by the previous diretions $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_{k-1}$. The following lemma describes how [Fletcher 75].

Lemma 4 *Assume that the error function is quadratic with constant Hessian $E''(\mathbf{w})$. Let the direction vectors be recursively defined by*

$$\mathbf{p}_k = \Leftrightarrow E'(\mathbf{w}_k) + \beta_{k-1}\mathbf{p}_{k-1} \tag{3.42}$$

where

$$\beta_{k-1} = \frac{|E'(\mathbf{w}_k)|^2 \Leftrightarrow E'(\mathbf{w}_k)^T E'(\mathbf{w}_{k-1})}{|E'(\mathbf{w}_{k-1})|^2}\ , \quad \beta_{-1} = 0. \tag{3.43}$$

Then the following three conditions hold

$$\begin{aligned} \mathbf{p}_k^T E''(\mathbf{w})\mathbf{p}_i &= 0\ , \quad i < k, && (3.44) \\ E'(\mathbf{w}_k)^T E'(\mathbf{w}_i) &= 0\ , \quad i < k, && (3.45) \\ \Leftrightarrow E'(\mathbf{w}_k)^T\mathbf{p}_k &= |E'(\mathbf{w}_k)|^2\ . && (3.46) \end{aligned}$$

The conditions are referred to as mutually conjugacy, orthogonal gradient and descent conditions.

Proof. We prove the lemma by induction. The initial step for $\mathbf{p}_0$ and $\mathbf{p}_1$ is easy to verify using the fact that $\mathbf{p}_0 = \Leftrightarrow E'(\mathbf{w}_0)$. We leave that to the reader. Assume that the lemma is true for all $i < k$. We first prove the orthogonal gradient condition. Using (3.40) and (3.42) we have

$$\begin{aligned} E'(\mathbf{w}_k)^T E'(\mathbf{w}_i) &= \left(E'(\mathbf{w}_{k-1}) + \alpha_{k-1}\mathbf{p}_{k-1}^T E''(\mathbf{w})\right)^T E'(\mathbf{w}_i) \\ &= E'(\mathbf{w}_{k-1})^T E'(\mathbf{w}_i) + \alpha_{k-1}\mathbf{p}_{k-1}^T E''(\mathbf{w})\left(\beta_{i-1}\mathbf{p}_{i-1} + \mathbf{p}_i\right). \end{aligned}$$

When $i < k \Leftrightarrow 1$, this is zero by (3.44) and (3.45), and when $i = k \Leftrightarrow 1$, it is zero by (3.39), (3.44) and (3.46). Thus, the orthogonal gradient condition (3.45) is true. Using (3.42) and (3.40)

$$\begin{aligned} \mathbf{p}_k^T E''(\mathbf{w})\mathbf{p}_i &= (\Leftrightarrow E'(\mathbf{w}_k) + \beta_{k-1}\mathbf{p}_{k-1})^T E''(\mathbf{w})\mathbf{p}_i \\ &= \Leftrightarrow\frac{1}{\alpha_i} E'(\mathbf{w}_k)\left(E'(\mathbf{w}_{i+1}) \Leftrightarrow E'(\mathbf{w}_i)\right) + \beta_{k-1}\mathbf{p}_{k-1}^T E''(\mathbf{w})\mathbf{p}_i \qquad (3.47) \end{aligned}$$

When $i < k \Leftrightarrow 1$, this is zero by (3.44) and (3.45), and when $i = k \Leftrightarrow 1$, it is zero by (3.39), (3.43), (3.45) and (3.46). So the mutually conjugacy condition is true. Finally, by (3.42) and (3.36)

$$\begin{aligned} \Leftrightarrow E'(\mathbf{w}_k)^T \mathbf{p}_k &= \Leftrightarrow E'(\mathbf{w}_k)^T \left(\Leftrightarrow E'(\mathbf{w}_k)^T + \beta_{k-1}\mathbf{p}_{k-1}\right) \\ &= E'(\mathbf{w}_k)^T E'(\mathbf{w}_k). \qquad (3.48) \end{aligned}$$

Thus, the descent condition is true, which ends the proof. □

Lemma 4 does not hold for non-quadratic functions. However, the direction vectors produced by lemma 4 will be approximately non-interfering, since a non-quadratic error function can be approximated by a quadratic as in (3.38). Based on lemma 4 the standard conjugate gradient method can be formulated as follows.

```
1. Select initial weight vector $\mathbf{w}_0$;
   $\mathbf{r}_0 = \Leftrightarrow E'(\mathbf{w}_0)$;
   $\mathbf{p}_0 = \mathbf{r}_0$;

2. $\alpha_k = \min_\alpha E(\mathbf{w}_k + \alpha\mathbf{p}_k)$;

3. $\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k\mathbf{p}_k$;
   $\mathbf{r}_{k+1} = \Leftrightarrow E'(\mathbf{w}_{k+1})$;

4. if ($k$ mod $N = 0$) then
      $\mathbf{p}_{k+1} = \mathbf{r}_{k+1}$ ;
   else
      $\beta_k = \frac{|\mathbf{r}_{k+1}|^2 - \mathbf{r}_{k+1}^T\mathbf{r}_k}{|\mathbf{r}_k|^2}$ ;
      $\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k$ ;

5. $k = k + 1$ ; terminate or go to 2.
```

The learning rate in step 2 is usually determined by a one dimensional line search, which can be very time consuming. The scaled conjugate gradient algorithm described in the end of this section and in appendix A, avoids this line search and estimates the learning rate by use of formula (3.39) and a scaling mechanism. If the conjugate gradient method is performed on a quadratic function, the method will terminate in at most N iterations, since by then, the whole weight space has been minimized, because of the non-interference condition. When the method is used on non-quadratic functions, this might not be the case, and the search direction is reset to the gradient descent direction. In practice, however, the method often terminates in $i \ll N$ iterations, also on non-quadratic functions.

3.2.2 Convergence rate

We now turn to the convergence rate of conjugate gradient methods. In order to be able to say anything about the convergence rate, we again have to assume that the error function is quadratic and additionally that the Hessian matrix is constant over time. We refer to the Hessian by $E^{''}(\mathbf{w})$. Thus, the bounds that we present, are not stricly valid for non-quadratic error functions, but merely approximations.

The convergence rate depends very much on the distribution of the eigenvalues of the Hessian matrix. If the eigenvalues fall into multiple or close groups, the convergence rate will be fast. This can be realized by the following. Using (3.42) and the relation

$$E^{'}(\mathbf{w}_{k+1}) \approx E^{'}(\mathbf{w}_k) + \alpha_k E^{''}(\mathbf{w})\mathbf{p}_k, \tag{3.49}$$

we see, that the conjugate direction vectors

$$\mathbf{p}_k \in K_k(E^{''}(\mathbf{w}), E^{'}(\mathbf{w}_0)), \tag{3.50}$$

where

$$K_k(E^{''}(\mathbf{w}), E^{'}(\mathbf{w}_0)) = span(E^{'}(\mathbf{w}_0), E^{''}(\mathbf{w})E^{'}(\mathbf{w}_0), \ldots, E^{''}(\mathbf{w})^{k-1}E^{'}(\mathbf{w}_0)) \tag{3.51}$$

is the *Krylov subspace* [Sluis and Horst 86]. The weight vectors, generated in the conjugate gradient algorithm, then have the folllowing property.

$$\mathbf{w}_k = \mathbf{w}_0 + P_k(E^{''}(\mathbf{w}))E^{'}(\mathbf{w}_0) \quad \in \quad \mathbf{w}_0 + K_k(E^{''}(\mathbf{w}), E^{'}(\mathbf{w}_0)), \tag{3.52}$$

where $P_k(E^{''}(\mathbf{w}))$ is a matrix polynomial of degree k. Corresponding to $E^{'}(\mathbf{w}_0)$ there are uniquely determined eigenvalues $\lambda_1 < \lambda_2 < \ldots < \lambda_m$ and normalized eigenvectors $\mathbf{e}_1, \ldots, \mathbf{e}_m$ of $E^{''}(\mathbf{w})$ such that

$$E^{'}(\mathbf{w}_0) = \sum_{i=1}^{m} \mu_i \mathbf{e}_i. \tag{3.53}$$

These eigenvalues and eigenvectors are the *active* ones. The possible other eigenvalues and eigenvectors do not participate in the conjugate gradient process. Obviously the maximum dimension of the Krylov subspace for increasing k is equal to m, and the conjugate gradient algorithm will terminate in m iterations. If the Hessian only has a few distinct eigenvalues, then m can be expected to be small.

As was the case for the gradient descent algorithm, the convergence rate can be bounded by the condition number of the Hessian. The following lemma states how.

Lemma 5 *Assume that the error function is quadratic with constant Hessian $E^{''}(\mathbf{w})$. At every step in the conjugate gradient algorithm it holds that*

$$\frac{E(\mathbf{w}_{k+1}) \Leftrightarrow E(\mathbf{w}_*)}{E(\mathbf{w}_k) \Leftrightarrow E(\mathbf{w}_*)} \leq 4 \left(\frac{\sqrt{\kappa} \Leftrightarrow 1}{\sqrt{\kappa} + 1} \right)^2,$$

where κ is the condition number of $E^{''}(\mathbf{w})$, and $\mathbf{w}_*$ is the minimum of the error.

Proof. We have by (3.5) that

$$E(\mathbf{w}_{k+1}) \Leftrightarrow E(\mathbf{w}_*) = \frac{1}{2}(\mathbf{w}_{k+1} \Leftrightarrow \mathbf{w}_*)^T E''(\mathbf{w})(\mathbf{w}_{k+1} \Leftrightarrow \mathbf{w}_*) \tag{3.54}$$

By (3.52) and since $E(\mathbf{w}_{k+1})$ minimizes the error over the Krylov subspace, we get

$$\begin{aligned} E(\mathbf{w}_{k+1}) \Leftrightarrow E(\mathbf{w}_*) &= \min_{P_{k+1}} \frac{1}{2}(\mathbf{w}_0 \Leftrightarrow \mathbf{w}_*)^T E''(\mathbf{w}) \\ &\quad \left(I + E''(\mathbf{w})P_{k+1}(E''(\mathbf{w}))\right)^2 (\mathbf{w}_0 \Leftrightarrow \mathbf{w}_*). \end{aligned} \tag{3.55}$$

The Hessian $E''(\mathbf{w})$ can be written in the form

$$E''(\mathbf{w}) = \sum_{i=1}^{N} \lambda_i \mathbf{e}_i \mathbf{e}_i^T, \tag{3.56}$$

where $\mathbf{e}_i$ and λ_i are the eigenvectors and corresponding eigenvalues of the Hessian. Similarly can the term $(\mathbf{w}_0 \Leftrightarrow \mathbf{w}_*)$ be written as

$$\mathbf{w}_0 \Leftrightarrow \mathbf{w}_* = \sum_{i=1}^{N} \xi_i \mathbf{e}_i, \tag{3.57}$$

with appropriate coefficients ξ_i. So equation (3.55) is

$$\begin{aligned} E(\mathbf{w}_{k+1}) \Leftrightarrow E(\mathbf{w}_*) &= \min_{P_{k+1}} \frac{1}{2} \left(\sum_{i=1}^{N} \xi_i \mathbf{e}_i\right)^T \left(\sum_{i=1}^{N} \lambda_i \mathbf{e}_i \mathbf{e}_i^T\right) \\ &\quad \left(I + \left(\sum_{i=1}^{N} \lambda_i \mathbf{e}_i \mathbf{e}_i^T\right) P_{k+1} \left(\sum_{i=1}^{N} \lambda_i \mathbf{e}_i \mathbf{e}_i^T\right)\right)^2 \left(\sum_{i=1}^{N} \xi_i \mathbf{e}_i\right) \\ &= \min_{P_{k+1}} \frac{1}{2} \left(\sum_{i=1}^{N} \lambda_i \xi_i \mathbf{e}_i\right)^T \left(I + \sum_{i=1}^{N} \lambda_i P_{k+1}(\lambda_i) \mathbf{e}_i \mathbf{e}_i^T\right)^2 \left(\sum_{i=1}^{N} \xi_i \mathbf{e}_i\right) \\ &= \min_{P_{k+1}} \frac{1}{2} \sum_{i=1}^{N} \left(1 + \lambda_i P_{k+1}(\lambda_i)\right)^2 \lambda_i \xi_i^2 \\ &\le \min_{P_{k+1}} \max_{\lambda \in [\lambda_{min}, \lambda_{max}]} \left(1 + \lambda P_{k+1}(\lambda)\right)^2 \frac{1}{2} \sum_{i=1}^{N} \lambda_i \xi_i^2 \\ &= \min_{P_{k+1}} \max_{\lambda \in [\lambda_{min}, \lambda_{max}]} \left(1 + \lambda P_{k+1}(\lambda)\right)^2 \left(E(\mathbf{w}_0) \Leftrightarrow E(\mathbf{w}_*)\right). \end{aligned} \tag{3.58}$$

Summarising equation (3.58), we have the following relation

$$\frac{E(\mathbf{w}_{k+1}) \Leftrightarrow E(\mathbf{w}_*)}{E(\mathbf{w}_0) \Leftrightarrow E(\mathbf{w}_*)} \le \min_{P_{k+1}} \max_{\lambda \in [\lambda_{min}, \lambda_{max}]} \left(1 + \lambda P_{k+1}(\lambda)\right)^2. \tag{3.59}$$

We now need to find a polynomial $Q_{k+1}(\lambda) = 1 + \lambda P_{k+1}(\lambda)$, that is small in the interval $[\lambda_{min}, \lambda_{max}]$, subject to the constraint that $Q_{k+1}(0) = 1$. Chebyhevs polynomials C_k, defined by

$$C_k(x) = \frac{1}{2}\left((x + \sqrt{x^2 \Leftrightarrow 1})^k + (x \Leftrightarrow \sqrt{x^2 \Leftrightarrow 1})^k\right) \tag{3.60}$$

are well suited for such problems [Golub and Loan]. They are small in the interval $[\Leftrightarrow 1, 1]$, but grow rapidly off this interval. As a consequence, the polynomial

$$Q_{k+1}(\lambda) = \frac{C_{k+1}\left((\lambda_{max} + \lambda_{min} \Leftrightarrow 2\lambda)/(\lambda_{max} \Leftrightarrow \lambda_{min})\right)}{C_{k+1}\left((\lambda_{max} + \lambda_{min})/(\lambda_{max} \Leftrightarrow \lambda_{min})\right)} \tag{3.61}$$

satifies $Q_0 = 1$ and tends to be small in the interval $[\lambda_{min}, \lambda_{max}]$. Using Chebychevs polynomials, we get

$$\begin{aligned} \frac{E(\mathbf{w}_{k+1}) \Leftrightarrow E(\mathbf{w}_*)}{E(\mathbf{w}_0) \Leftrightarrow E(\mathbf{w}_*)} &\leq \max_{\lambda \in [\lambda_{min}, \lambda_{max}]} \left(Q_{k+1}(\lambda)\right)^2 \qquad (3.62) \\ &= \max_{\lambda \in [\lambda_{min}, \lambda_{max}]} \left(\frac{C_{k+1}\left((\lambda_{max} + \lambda_{min} \Leftrightarrow 2\lambda)/(\lambda_{max} \Leftrightarrow \lambda_{min})\right)}{C_{k+1}\left((\lambda_{max} + \lambda_{min})/(\lambda_{max} \Leftrightarrow \lambda_{min})\right)}\right)^2 \\ &= \left(\frac{1}{C_{k+1}\left((\lambda_{max} + \lambda_{min})/(\lambda_{max} \Leftrightarrow \lambda_{min})\right)}\right)^2 \\ &\leq 4\left(\frac{\sqrt{\kappa} \Leftrightarrow 1}{\sqrt{\kappa} + 1}\right)^{2(k+1)} . \end{aligned}$$

The desired result is now easily obtained from the last inequality. □

Lemma 5 predicts only linear convergence by a fixed factor $\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^2$. In order to state something about the so-called superlinear convergence behaviour, it is necessary to look further in to the distribution of the eigenvalues. We refer to [Sluis and Horst 86] for a more detailed evaluation of the convergence behaviour of conjugate gradient methods.

3.2.3 Scaled conjugate gradient

The scaled conjugate gradient algorithm (SCG) is described in detail in appendix A or [Møller 93a]. For that reason, we will only briefly describe the main ideas in the algorithm here. We will, however, discuss various possible improvements of the algorithm.

The estimation of the learning rate in the standard conjugate gradient algorithm is done with a line search routine. A line search performs a one dimensional iterative search for a learning rate in the direction of the current search direction. There are several drawbacks by doing a line search. It introduces new problem-dependent parameters, e.g., a parameter to determine how many iterations to perform before termination. Kinsella has shown, that this can have a major impact on the performance of conjugate gradient algorithm [Kinsella 92]. Furthermore, the line search does in each iteration involve several calculations of the error and/or the derivative to the error, which is time consuming.[5] SCG substitutes the line search by a scaling of the step that depends on success in error reduction and goodness of the quadratic approximation to the error. The algorithm encorporates ideas from the *model-trust region* methods in optimization and "safety" procedures that are absent in standard conjugate gradient.

If the error function would be strictly quadratic with a positive definite Hessian matrix, the learning rate given by formula (3.39) would be optimal. Using this formula for non-quadratic error functions, however, causes problems, because the Hessian matrix can be indefinite and the quadratic approximation to the error might not always be good. The

[5] Each calculation costs $O(PN)$ time.

key idea of SCG consists of the introduction of a scalar, λ_k, which is used to regulate the positive definiteness of the Hessian.[6] The Hessian is substituted with

$$E''(\mathbf{w}_k) + \lambda_k I, \tag{3.63}$$

so that the learning rate is given by

$$\alpha_k = \frac{\Leftrightarrow E'(\mathbf{w}_k)\mathbf{p}_k}{\mathbf{p}_k^T E''(\mathbf{w}_k)\mathbf{p}_k + \lambda_k|\mathbf{p}_k|^2} \tag{3.64}$$

λ_k is in each iteration raised or lowered according to how good the second order approximation is to the real error. The parameter Δ_k that measures the ratio between the real error change and the predicted quadratic error change is given by

$$\Delta_k = \frac{\left(\mathbf{p}_k^T E''(\mathbf{w}_k)\mathbf{p}_k + \lambda_k|\mathbf{p}_k|^2\right)\left(E(\mathbf{w}_k) \Leftrightarrow E(\mathbf{w}_{k+1})\right)}{\left(\Leftrightarrow E'(\mathbf{w}_k)^T\mathbf{p}_k\right)^2} \, . \tag{3.65}$$

An increase or decrease of the scaling parameter λ_k is controlled by the value of Δ_k. The term $E''(\mathbf{w}_k)\mathbf{p}_k$ in (3.64) can either be approximated by a one sided difference equation of the form

$$E''(\mathbf{w}_k)\mathbf{p}_k \approx \frac{E'(\mathbf{w}_k + \sigma_k\mathbf{p}_k) \Leftrightarrow E'(\mathbf{w}_k)}{\sigma_k} \; , \quad \sigma_k = \frac{\epsilon}{|\mathbf{p}_k|^2} \; , \;\; 0 < \epsilon \ll 1 \tag{3.66}$$

or calculated exactly as described in appendix C, [Møller 93c] or [Pearlmutter 93]. The exact calculation is only a bit more complicated than the approximation and both schemes costs $O(PN)$ time, where P is the number of patterns in the training set and N is the number of weights.

We will now turn to a discussion of possible improvements of the algorithm. The formula (3.64) for the learning rate can be viewed as a *1-step Newton line search estimation* [Møller 90a]. A j-step Newton estimation is defined as

$$\alpha_j = \alpha_{j-1} + \frac{\Leftrightarrow E'(\mathbf{w}_k + \alpha_{j-1}\mathbf{p}_k)\mathbf{p}_k}{\mathbf{p}_k^T E''(\mathbf{w}_k + \alpha_{j-1}\mathbf{p}_k)\mathbf{p}_k + \lambda_k|\mathbf{p}_k|^2} \; , \quad \alpha_0 = 0. \tag{3.67}$$

Using (3.67) in SCG with j as a user dependent parameter, would be to introduce a procedure similar to the line search, that SCG was designed to avoid. However, it might be better to set j to another constant than $j = 1$. To check if this is the case, a series of expriments were run on the two-spirals problem [Lang and Witbrock 89] with various values of j. A series of 10 runs were tested for each value of j, and the runs were terminated when all the patterns were classified correctly within a margin of 0.8.[7] The results are illustrated in table 3.1. We observe that the convergence with respect to the number of epochs improves with increasing j, except for $j = 3$. The convergence does, however, not improve enough to justify the additional computation time used in each iteration.

Peter Williams has suggested several interesting improvements to the SCG algorithm [Williams 91]. One is an improvement of the update formula of the scaling parameter λ_k, which is included in appendix A and [Møller 93a]. Another is an adaptive scheme

[6] This is essentially a Levenberg-Marquardt approach [Fletcher 75]

[7] The *exponential* error function described in appendix E was used in these experiments.

j	<epoch>	<cu>	failures
1	1052	4208	1
2	804	4824	0
3	840	6720	1
4	732	7320	0

Table 3.1: Average of 10 runs on the two-spirals problem with various values of the j parameter. The *<cu>* part is the average number of *complexity units*, which is an abstract measure that also takes the computational costs per epoch into account (one <cu> is equivalent to one forward or one backward propagation of all patterns in the training set).

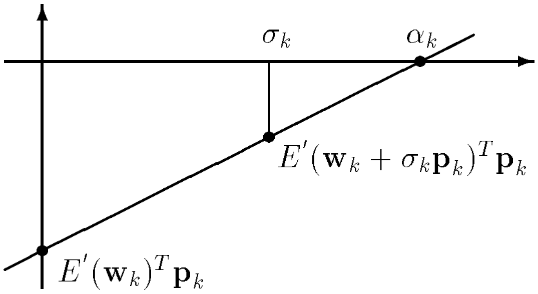
σ_k
α_k
$E'(\mathbf{w}_k + \sigma_k \mathbf{p}_k)^T \mathbf{p}_k$
$E'(\mathbf{w}_k)^T \mathbf{p}_k$

Figure 3.3: The extrapolation used to determine α_k by the one sided differencing scheme.

for the setting of the parameter ϵ in the one sided difference formula (3.66). We will present his findings here. In [Møller 93a] it was experimentally shown that the value of ϵ was not problem dependent as long as it was set to a small value. Williams argues, however, that a small gain in convergence can be achieved by adapting ϵ. If the error was strictly quadratic, the one sided differencing would be an exact calculation of $E''(\mathbf{w}_k)\mathbf{p}_k$ no matter the value of ϵ. So in this case there is no particular advantage in having $\epsilon \ll 1$. When the error is non-quadratic, Williams argue that the value of ϵ does not matter either. When the scaling parameter $\lambda_k \approx 0$ the learning rate calculation is equivalent to fitting a straight line to $E'(\mathbf{w}_k)^T \mathbf{p}_k$ and $E'(\mathbf{w}_k + \sigma_k \mathbf{p}_k)^T \mathbf{p}_k$ to give α_k as the estimated zero crossing of the line (see figure 3.3). The ideal σ_k would be one for which $\sigma_k = \alpha_k$. Note that α_k is the new learning rate that we want to estimate. This suggest a way of adapting ϵ. One adaptation rule given by Williams is

$$\epsilon_{k+1} = \sigma_k \left(\frac{\alpha_k}{\sigma_k} \right)^{\pi} \ , \quad 0 \leq \pi \leq 1. \tag{3.68}$$

Choosing $\pi = 0$ is equivalent to using the initial value ϵ_0 throughout. Non-zero values of π adapt ϵ_k so that, on average, the values of α_k and σ_k tend to be equalized. Note that if ϵ is approximately equal to the expected learning rate, then a finite difference estimate may provide a more faithful picture of the error surface than an extrapolated local quadratic model, such as the exact calculation of $E''(\mathbf{w}_k)\mathbf{p}_k$, especially if the quadratic approximation of the error is bad. In any case, Williams reports a minor speedup in convergence using this adaptation scheme. In appendix C, however, the author finds that the exact calculation of $E''(\mathbf{w}_k)\mathbf{p}_k$ in average yields the fastest convergence.

3.2.4 Stochastic conjugate gradient

In section 3.1 we saw, that the gradient descent algorithm could be used in two different modes, *on-line* and *off-line*. Conjugate gradient algorithms are in their standard form only applicable in off-line mode, because of the near optimal choice of learning rate in each iteration. In off-line mode the amount of computation time used in each iteration is dependent of the number of patterns in the training set. When the training set is large and contains a lot of redundant information this leads to redundant computations in an off-line algorithm. So although the convergence rate of conjugate gradient algorithms with respect to the number of iterations used is much better than that of gradient descent algorithms, it can often be observed that on-line gradient descent beats more sophisticated second-order algorithms on large redundant problems. The second-order algorithms "drown" in their own computations. It is however, possible to make stochastic versions of second-order algorithms, that updates weights based on smaller subsets of the training set. In appendix B and [Møller 93b] a stochastic version of the scaled conjugate gradient algorithm is described. We summarise this algorithm here. In section 3.4 on-line and off-line techniques are discussed in more detail.

The approach of updating the weights based on smaller subsets of data has been explored by several researchers. Kramer and Sangiovani-Vincentelli describes a two stage scheme, where the first stage involves training the network on the current subset of data until convergence and the second stage involves picking new patterns to successively increase the training subset [Kramer et al. 88]. Haffner *et al.* describes a similar scheme [Haffner et al. 88]. A problem with such schemes is that the network tends to get overspecialized when trained to convergence on small subsets of data. The training should be terminated and the subset increased when overspecialization occures. This point is, however, not easy to detect. Kuhn and Herzberg describes a scheme combined with conjugate gradient applied to a speech recognition problem, where the subset size is proportional to the number of output classes [Kuhn and Herzberg 90]. Similar for these schemes is, that the size of the data subsets and which patterns to include in them, is determined in a very *ad hoc* fashion, e.g., taking one pattern from each output class or adding a predetermined and constant number of patterns to the current subset. Furthermore, the schemes do not consider any validation of the size and patterns chosen, which means that the training process might diverge without detection.

Using standard sampling techniques as described in [Cochran 77], it is possible to define schemes that can validate each update and base the size of the data subset on these validations. We call such schemes *update-validation*. One such scheme was described in appendix B and [Møller 93b], where it was combined with the scaled conjugate gradient algorithm. The scheme is illustrated in figure 3.4. The scheme involves two blocks (subsets) of data, an *update* block and a *sample* block. An update of the weights is based on the update block and the sample block is used to validate each update. This validation involves the calculation of an *update probability* $\hat{P}_i$, that is an estimate of the probability that an update will decrease the error on the whole training set. This probability can be estimated under the assumption that the error of blocks of data is normaly distributed around the error of the whole training set, and then selecting the sample block as a simple random sample. The update probability can then be estimated by

$$\hat{P}_i = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\frac{\Delta\hat{\mu}_i}{\sigma_i}} e^{-\frac{1}{2}t^2} dt \ , \tag{3.69}$$

Figure 3.4: The update-validation scheme used in appendix B. $\hat{P}_i$ is an estimated probability that an update based on the current subset of data will decrease the error on the whole training set. The weights are updated with probability $\hat{P}_i$.

where $\hat{\mu}_i$ is the change in error of the current sample block before and after a possible update, and σ_i^2 is the error variance of the sample block. The size of the update block is optimized by means of a binary search method. The size of the sample block is chosen such that the error of a sample block is very close to the error of the training set with a high probability.

Only a little change in conjugate gradient algorithms is needed in order to be able to update on subsets of data of maybe varying size. The error function has to be normalized so that we operate on average error rather than total error. If the error is normalized then the error on subsets of data will be approximations to the error on the whole training set. The better the approximations are the better and more reliable will a conjugate gradient algorithm converge. Combining the above update-validation scheme with the scaled conjugate gradient algorithm yields good results as concluded in appendix B and [Møller 93b]. The major advantage of combining second-order training algorithms with stochastic schemes is, that a minimum is quickly localized because of more frequent updates, and the convergence is fast down into this minimum because of the second order properties of the algorithms.

As discussed later in section 3.4, it might be possible to improve the above update-validation scheme in various ways. One idea worth mentioning here also is the possible use of *active data selection* techniques to determine appropriate update blocks. Rather than selecting patterns by random sampling, the training could be made more efficient by selecting patterns that maximizes the information content of the update block.

3.3 Newton related methods

In this section we briefly describe the Newton method and some of its variations that are relevant in a neural network context. We conclude with a decription of the *one-step memoryless BFGS* method, which operates in $O(N)$ time and is very similar to the conjugate gradient methods [Battiti 92]. The Newton method is in its original form not applicable for training of neural networks, because it involves an inversion of the Hessian matrix, which is computational expensive. The Newton search direction $\mathbf{p}_k$ is defined by the linear system

$$E^{''}(\mathbf{w}_k)\mathbf{p}_k = \Leftrightarrow E^{'}(\mathbf{w}_k) \; , \tag{3.70}$$

which originates when minimizing the quadratic approximation to the error function. If the error is strictly quadratic and the Hessian is positive definite this algorithm converges in one iteration, starting from any initial weight vector. For non-quadratic error functions, the algorithm converges quadratically if the Hessian is positive definite and sufficiently close to the desired minimum. Whenever the Hessian is indefinite or ill-conditioned severe problems arise, such as too large learning rates and numerical problems. Many modified Newton algorithms exist that incorporate techniques to ensure a sufficiently positive definite and non-singular Hessian matrix. A class of such algorithms is the *quasi-Newton* methods. Quasi-Newton methods are based on the idea of accumulating curvature information as the iterations proceed, using the observed behaviour of $E(\mathbf{w}_k)$ and $E^{'}(\mathbf{w}_k)$. The methods builds up an approximation of the Hessian or to the inverse of the Hessian. In the following we describe how to approximate the inverse of the Hessian. In the beginning of the kth iteration of a quasi-Newton method, an approximate inverse Hessian matrix H_k is available, which is intended to reflect the inverse curvature information already accumulated. The search direction $\mathbf{p}_k$ is then computed from the Newton formula

$$\mathbf{p}_k = \Leftrightarrow H_k E^{'}(\mathbf{w}_k). \tag{3.71}$$

The initial matrix H_0 is usually taken to be the identity matrix, so that the first iteration is equivalent with a gradient descent update. H_k is then updated recursively in each iteration so that it approximates the inverse curvature along the current search direction $\mathbf{p}_k$. If we assume that the error function is quadratic, then we have by (3.40) that

$$E^{''}(\mathbf{w}_k)^{-1}\left(E^{'}(\mathbf{w}_{k+1}) \Leftrightarrow E^{'}(\mathbf{w}_k)\right) = \alpha_k \mathbf{p}_k^T = (\mathbf{w}_{k+1} \Leftrightarrow \mathbf{w}_k). \tag{3.72}$$

Based on (3.72), H_{k+1} is required to satisfy the so-called *quasi-Newton condition*

$$H_{k+1}\left(E^{'}(\mathbf{w}_{k+1}) \Leftrightarrow E^{'}(\mathbf{w}_k)\right) = (\mathbf{w}_{k+1} \Leftrightarrow \mathbf{w}_k). \tag{3.73}$$

There are several ways to update H_k in order to satisfy this condition. We will not describe these in detail here, but only mention the one that leads us to the one-step memoryless BFGS formula. It is generally believed that the most effective formula is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) formula given by

$$H_{k+1} = H_k + \left(1 + \frac{\mathbf{y}_k^T H_k \mathbf{y}_k}{\mathbf{y}_k^T \mathbf{s}_k}\right)\frac{\mathbf{s}_k \mathbf{s}_k^T}{\mathbf{s}_k^T \mathbf{y}_k} \Leftrightarrow \frac{\mathbf{s}_k \mathbf{y}_k^T H_k + H_k \mathbf{y}_k \mathbf{s}_k^T}{\mathbf{y}_k^T \mathbf{s}_k} \; , \tag{3.74}$$

where $\mathbf{y}_k = \left(E^{'}(\mathbf{w}_{k+1}) \Leftrightarrow E^{'}(\mathbf{w}_k)\right)$ and $\mathbf{s}_k = (\mathbf{w}_{k+1} \Leftrightarrow \mathbf{w}_k)$. Formula (3.74) ensures that H_k is symmetric and positive definite. The BFGS algorithm involves storage of a $N \times N$ matrix and $O(PN^2)$ in computation requirements. So in its current form, the BFGS algorithm is not applicable to training of neural networks, at least not on large networks. It is, however, possible to simplify formula (3.74) so that the time and memory requirements are $O(PN)$ and $O(N)$ respectively. The idea is to apply the BFGS formula to the identity matrix I, rather than to H_k. Thus H_{k+1} is determined without reference to the previous

H_k, and hence the update procedure is *memoryless.* By (3.71) and setting $H_k = I$ in (3.74), the search direction is given by

$$\begin{aligned}\mathbf{p}_{k+1} &= \Leftrightarrow E'(\mathbf{w}_{k+1}) \Leftrightarrow \left(1 + \frac{\mathbf{y}_k^T \mathbf{y}_k}{\mathbf{y}_k^T \mathbf{s}_k}\right) \frac{\mathbf{s}_k^T E'(\mathbf{w}_{k+1}) \mathbf{s}_k}{\mathbf{s}_k^T \mathbf{y}_k} \\ &\quad + \frac{\mathbf{y}_k^T E'(\mathbf{w}_{k+1}) \mathbf{s}_k + \mathbf{s}_k^T E'(\mathbf{w}_{k+1}) \mathbf{y}_k}{\mathbf{y}_k^T \mathbf{s}_k}. \end{aligned} \qquad (3.75)$$

The one-step memoryless BFGS method is usually combined with a line search to estimate appropriate learning rates. See [Battiti 89] for further reading about this. When exact line searches are made, i.e., an optimal learning rate is found in each iteration, the algorithm is the same as the conjugate gradient algorithm. This is easily verified by using the observation that $\mathbf{p}_k^T E'(\mathbf{w}_{k+1}) = 0$, which implies that $\mathbf{p}_k^T \mathbf{y}_k = \Leftrightarrow \mathbf{p}_k^T E'(\mathbf{w}_k)$. The algorithm is considered to have similar convergence properties as the conjugate gradient algorithms. This is also consistent with results reported by Battiti, who concludes that SCG and the one-step memoryless BFGS yields comparable results [Battiti 92].

3.4 On-line versus off-line discussion

As mentioned in section 3.1, training methods of feed-forward neural networks can roughly be divided up into two categories, *on-line* and *off-line* techniques. There is some confusion about the terms "on-line" and "off-line". The term "on-line" refers historically to methods where the weights are updated based only on information from one single pattern, while "off-line" refers to methods where the weights are updated based on information from the whole training set. We will use the "on-line" term in a broader sense, referencing it to methods that update weights *independent* of the training set size. Some researchers also use the terms *stochastic* and *batch* as alternative names for on-line and off-line. Several examples of both on-line and off-line methods have been described in the past sections. The gradient descent method can be used in both on-line and off-line mode. The conjugate gradient algorithms and other second order algorithms are in standard form all off-line algorithms, but can also with some modifications be used in on-line mode as described in section 3.2.4. Off-line schemes described in the past sections include all the adaptive learning rate schemes in section 3.1.4, the quickprop method in section 3.1.6 and scaled conjugate gradient in 3.2.3. Methods that also apply to on-line mode are the learning rate schedules in section 3.1.5, the optimal learning rate estimation and reduction of large curvature components in section 3.1.7, and the stochastic conjugate gradient scheme in 3.2.4.

There are drawbacks and benefits with both types of update schemes. Off-line algorithms are easier to analyse concerning convergence properties, they can choose an optimal learning rate in each iteration and can yield very high accuracy solutions. They suffer, however, from the fact that the time to prepare a weight update increases with increasing training set size. This turns out to be crucial on many large scale problems. On-line methods can be used when the patterns are not available before training starts, and a continuous adaptation to a stream of input-output relations is desired. The randomness in the updates can help escape local minima and the time to prepare a weight update is not affected by increasing training set size. On-line methods are not good to produce

Figure 3.5: The search direction of off-line algorithms is an accumulation of partial search directions. If redundancy is present, a sum of a small fraction of these partial directions might produce a direction $\hat{\mathbf{p}}_k$, that is "close" to the desired search direction $\mathbf{p}_k$, i.e., within a small neighborhood of the desired direction.

high accuracy solutions, e.g, function approximation, and the setting of the learning rate is not well understood.

Off-line methods like standard conjugate gradient or quasi-Newton methods should in theory yield the fastest convergence. This is, however, not the case on large scale problems that are characterized by large and very redundant training sets. The problem that arises is illustrated in figure 3.5. If many of the patterns in the training set possess redundant information the contributions to the search direction will be similar, and waiting for all contributions before updating can be a waste of time. There is obviously a tradeoff between the accuracy of the search direction and the computational costs to calculate it. In other words, redundancy in training sets produces redundant computations in off-line algorithms.

In addition to the description of the stochastic scaled conjugate gradient algorithm, appendix B and [Møller 93b] presents a method of measuring the amount of redundancy in training sets. We will summarize these redundancy aspects here. First it is important to recognize that the redundant computations in an off-line algorithm cannot be expected to be a constant even though the redundant information in the training set is constant. The redundant computations will also depend on the network dynamics, i.e., of the current weights. So a measure of redundancy of training sets alone cannot be expected to give sufficient information about the amount of redundant computations made by an off-line algorithm. Such a measure could, however, give a first estimate of what to expect of the training process. A measure of redundancy can be based on the standard information theory [Shannon and Warren 64]. We define a redundancy measure for classification problems with M different output classes and P discrete input vectors of length L, each attribute having V possible values. The *Conditional Population Entropy* (CPE) is defined as

$$CPE = \Leftrightarrow \sum_{m=1}^{M} p(c_m) \sum_{l=1}^{L} \sum_{v=1}^{V} p(x_v^l | c_m) \log p(x_v^l | c_m) \ , \qquad (3.76)$$

where $p(c_m)$ is the probability that an input vector belongs to the mth class and $p(x_v^l|c_m)$ is the probability that the lth attribute of an input vector x has value v given that x belongs to the mth class. If we imagine that the whole training set is split up into M disjoint sets, one for each class, then CPE is the average information content of these sets. We can also interprete CPE as the average number of bits needed to code one input pattern given knowledge about the classes. Clearly the value of CPE states something about the degree of similarity between the input patterns. If CPE is small, only a small number of bits is needed to code the input patterns, and hence there must be great similarity between the patterns. Based on these observations, a redundancy measure RE can be defined as

$$RE = \frac{\log V \Leftrightarrow \frac{CPE}{L}}{\log V} \ , \tag{3.77}$$

where $\log V$ is the necessary number of bits needed to code one attribute if all values are equally likely and $\frac{CPE}{L}$ is the average number of bits needed to code one attribute. One drawback with this redundancy measure is that it does not take correlations between attributes into account. Nevertheless, in appendix B and [Møller 93b] it is found that there is a strong correlation between the value of RE and the efficiency of various off-line and on-line algorithms, when trained on the particular problems. The measure can be expanded to work on non-classification problems also by splitting the input and/or the output ranges up in a set of discrete intervals.

The neural network community seems to be split up into two blocks regarding the question of which direction research should go concerning on-line and off-line algorithms. Since many practical neural network problems are characterized by large redundant training sets, some researchers think that the effort should be put into finding better speed up techniques to apply with on-line gradient descent. One such approach is the on-line estimation of learning rate and reduction of large curvature components by Le Cun *et al.* described in section 3.1.7. This is a very promising approach and should be investigated further. The other group, to which the author belongs, believes that the power of second order methods can and should be used also on large scale problems. The approach is to make the second order methods stochastic, i.e., update on smaller blocks of data. The general approach can be characterized as follows.

- Accumulate error, gradient and/or Hessian information for a length of time that is adaptively chosen. The time interval should be large enough to ensure safe weight updates with near optimal learning rates but small enough to avoid redundant computations.

- Use a second order algorithm like SCG to update the weights.

One such approach is the stochastic SCG algorithm (SSCG) described in section 3.2.4. In SSCG a *validation scheme* is defined to validate each update. This scheme involves random sampling of additional patterns that are not used in the current update process. This should *in principle* not be necessary, since enough information should already be available in the data used to prepare an update. If the data is redundant, the estimated search directions produced by accumulation of partial directions will converge.[8] This means that the angle θ_k between the estimated search direction and the real direction

[8] We assume here that we have an infinite stream of data available.

(see figure 3.5) converges to zero. It might be possible to define an adaptive scheme that involves the convergence of θ_k to determine when to stop collecting new patterns. A problem here is that the convergence of θ_k will not necessarily be monotonic, so the scheme has to take small fluctuations of θ_k into account. These ideas is a subject for further research.

Another approach to solve the problem with redundant data is to control what data is used in training, this is often referred to as *active data selection.* This approach could be used to improve the SSCG method and other second order stochastic methods. Rather than selecting data by random sampling, the training could be made more efficient by actively selecting data, that maximizes the information density of the training subset. Active data selection has been extensively studied in economic theory and statistics, see for example [Fedorov 72]. In a neural network context Plutowski *et al.* and MacKay have recently proposed different schemes to select data [Plutowski et al.], [MacKay 92]. We shortly summarize.

Plutowski *et al.* considers the problem of selecting training subsets from a large noise-free data set [Plutowski et al.]. They assume that a large amount of data has already been gathered, and work on principles for selecting a subset of data for efficient training. A drawback with the method is that the entire data set has to be examined in order to decide which example to add to the current training subset. Plutowski *et al.* reports, however, an order of magnitude faster convergence than if the network was trained on the entire data set. Plutowski *et al.* uses in the training process the least mean square error function

$$E(\mathbf{w}) = \frac{1}{2n}\sum_{p=1}^{n}\left(g(\mathbf{x}_p) \Leftrightarrow f(\mathbf{x}_p,\mathbf{w})\right)^2 \ , \tag{3.78}$$

where $(\mathbf{x}_p, g(\mathbf{x}_p))$ is the pth input-output relation and $f(\mathbf{x}_p,\mathbf{w})$ is the network output on $\mathbf{x}_p$.[9] A criterion for selecting training examples that works well in conjunction with the error function used for training is the *Integrated Squared Bias* (ISB) given by

$$ISB(\mathbf{X}_n) = \int \left(g(\mathbf{x}) \Leftrightarrow f(\mathbf{x},\mathbf{w}_n)\right)^2 \mu(\mathbf{dx}) \ , \tag{3.79}$$

where $\mathbf{X}_n$ is a data subset of size n, $\mathbf{w}_n$ is the set of weights that minimizes (3.78) on $\mathbf{X}_n$, and μ is a distribution over the inputs. Clearly, finding a subset $\mathbf{X}_n$ that minimizes (3.79) gives us a subset representative of the whole data set. Finding such a subset is computational impractical. Plutowski *et al.* approximates a solution by successively adding new examples $\mathbf{x}_{n+1}$ to the training subset so as to maximize the decrement in ISB given by $\triangle ISB(\mathbf{x}_{n+1}|\mathbf{X}_n) = ISB(\mathbf{X}_n) \Leftrightarrow ISB(\mathbf{X}_n \cup \{\mathbf{x}_{n+1}\})$. Using first order Taylor expansions this decrement can be approximated by

$$\begin{aligned}
\triangle ISB(\mathbf{x}_{n+1}|\mathbf{X}_n) \;&\approx\; \left((g(\mathbf{x}_{n+1}) \Leftrightarrow f(\mathbf{x}_{n+1},\mathbf{w}_n))f'(\mathbf{x}_{n+1},\mathbf{w}_n)^T E''(\mathbf{X}_n,\mathbf{w}_n)^{-1}\right)^T \\
&\quad \sum_{p=1}^{P} f'(\mathbf{x}_p,\mathbf{w}_n)(g(\mathbf{x}_p) \Leftrightarrow f(\mathbf{x}_p,\mathbf{w}_n)) \qquad (3.80) \\
&= \left(E'(\mathbf{x}_{n+1},\mathbf{w}_n)^T E''(\mathbf{X}_n,\mathbf{w}_n)^{-1}\right)\sum_{p=1}^{P} E'(\mathbf{x}_p,\mathbf{w}_n)
\end{aligned}$$

[9] We assume w.l.g. that the network has one output unit.

where the sum runs over patterns in the whole data set. This approximation depends on the weight update rule. Formula (3.80) is derived when using a Newton weight update rule. Because the mean least square error function is used, the Hessian can be approximated by $E^{'}(\mathbf{X}_n, \mathbf{w}_n)E^{'}(\mathbf{X}_n, \mathbf{w}_n)^T$. A similar rule could also be derived for gradient descent or conjugate gradient. These rules would be more simple and save computation, but also not as accurate. From the last term in (3.80) we see that maximizing $\triangle ISB$ is equivalent to picking the example having individual error gradient most highly correlated with the error gradient of the entire data set.

It is possible to simplify (3.80) even more by ignoring all network gradient information. Interestingly, we then end up with a *maximum error* criterion selecting examples with maximum network error. This criterion is much cheaper to compute than (3.80) and works at least on some test problems in [Plutowski et al.] as well as the original criterion. It is, however, not clear whether the network gradient information can be ignored in general. Plutowski *et al.* also show that selecting examples by the ISB criterion works better than straightforward random sampling, which suggests that the validation scheme described in section 3.2.4 could be improved by exchanging the random sampling with a more sophisticated approach. Such an approach should, however, be independent of the size of the data set, so the ISB approach would need to be "relaxed" somehow in order to be usable.

MacKay uses a different approach than Plutowski *et al.* including noise in his model, but finally ends up with a similar result. He uses a Bayesian perspective to obtain an information based criterion about what example to pick next. The *posterior probability* of the weights $P(\mathbf{w}|X, \aleph)$ can in Bayesian terms be expressed as

$$P(\mathbf{w}|\mathbf{X}, \aleph) = \frac{P(\mathbf{X}|\mathbf{w}, \aleph)P(\mathbf{w}|\aleph)}{P(\mathbf{X}|\aleph)} \ , \tag{3.81}$$

where $\mathbf{X}$ is a subset of data and $\aleph$ is the network. The normalizing term $P(\mathbf{X}|\aleph)$ is a constant and can be ignored in what follows. $P(\mathbf{X}|\mathbf{w}, \aleph)$ is often denoted the *sample likelihood* and is associated with the error of the data and $P(\mathbf{w}|\aleph)$ is the *prior probability* of the weights, which is associated with regularization functions, such as weight decay [Weigend et al. 90]. The sample likelihood and the prior are defined as

$$\begin{aligned} P(\mathbf{X}|\mathbf{w}, \aleph) &= \exp(\Leftrightarrow\beta E_D(\mathbf{w})) \qquad (3.82) \\ P(\mathbf{w}|\aleph) &= \exp(\Leftrightarrow\alpha E_w(\mathbf{w})) \ , \end{aligned}$$

where β is the inverse of a noise parameter, $E_D(\mathbf{w})$ is an error function on the form (3.78), α is a regularization parameter, and $E_w(\mathbf{w})$ is a regularization function, such as weight decay. If we define $M(\mathbf{w})$ to be

$$M(\mathbf{w}) = \beta E_D(\mathbf{w}) + \alpha E_w(\mathbf{w})) \ , \tag{3.83}$$

then the posterior probability is given by

$$P(\mathbf{w}|\mathbf{X}, \aleph) = \exp(\Leftrightarrow M(\mathbf{w})) \ , \tag{3.84}$$

Let $\mathbf{X}_n$ be a subset of examples of size n. Then $P(\mathbf{w}|\mathbf{X}_n, \aleph)$ is the posterior probability when these n examples are used in training. The idea now, is to construct a measure of the information gained by adding a new example to the subset. Such a measure can be constructed by means of entropy functions. The entropy S_n of $P(\mathbf{w}|\mathbf{X}_n, \aleph)$ is defined as

$$S_n = \Leftrightarrow \int P(\mathbf{w}|\mathbf{X}_n, \aleph) \log P(\mathbf{w}|\mathbf{X}_n, \aleph) \mathbf{dw} \ . \tag{3.85}$$

The higher the value of S_n the more "uncertainty" is accociated with the weights. See for example [Gallager 68] for further reading about entropy functions. The information gained by adding a new example $\mathbf{x}_{n+1}$ can now be expressed as the change in entropy $\triangle S_n = S_n \Leftrightarrow S_{n+1}$.

Let $\mathbf{w}_n$ denote the weights that minimizes $M(\mathbf{w})$, i.e., maximizes $P(\mathbf{w}|\mathbf{X}_n, \aleph)$, on the subset $\mathbf{X}_n$. MacKay approximates the information gain by using a quadratic approximation of $M(\mathbf{w})$ expanded around $\mathbf{w}_n$, and a first order approximation of the new Hessian matrix $M''(\mathbf{w}_{n+1})$ from $M''(\mathbf{w}_n)$. Based on these somewhat crude approximations we finally get the formula

$$\triangle S_n \approx \frac{1}{2} \log \left(1 + \beta f'(\mathbf{x}_{n+1}, \mathbf{w}_n)^T M''(\mathbf{w}_n)^{-1} f'(\mathbf{x}_{n+1}, \mathbf{w}_n)\right) \ . \tag{3.86}$$

If we compare this formula with formula (3.80) for $\triangle ISB$, we observe that there is a great similarity. The only main difference is that $\triangle ISB$ involves error gradients of the new example while $\triangle S_n$ involves network gradients. Nevertheless, since the term $f'(\mathbf{x}_{n+1}, \mathbf{w}_n)^T M''(\mathbf{w}_n)^{-1} f'(\mathbf{x}_{n+1}, \mathbf{w}_n)$ can be interpreted as the variance of the network when example $\mathbf{x}_{n+1}$ is sampled, we obtain the maximum information gain by picking the example with the largest error. This is consistent with Plutowski *et al.*'s results. MacKay generalizes these results to selecting examples in specific regions of input space an selecting multiple examples. MacKay emphasizes, however, that some care should be taken when applying these techniques. The information gain estimates the utility of a data example assuming that the network model is correct, i.e., that the network is able to implement the desired and usually unknown input-output mapping. If the model is actually an approximation then the method might lead to undesirable results. For further reading about the Bayesian approach see [Buntine and Weigend 91b], [MacKay 91a] and [MacKay 91b].

3.5 Conclusion

Training algorithms for feed-forward neural networks can roughly be divided up in two categories, gradient descent algorithms and second-order algorithms, like conjugate gradient. There is up to date no simple answer to the question of which algorithm to prefer in general. Gradient descent exhibits linear convergence, but can, nevertheless, converge faster than second-order algorithms, when used in on-line mode and redundancy is present in the data. The author recognizes two approaches, that should be explored further

- Improve the on-line gradient descent techniques. The work of Darken *et al.* and Le Cun *et al.* in section 3.1.5 and 3.1.7 respectively are promising approaches in this area.

- Develop stochastic versions of standard second-order methods, that can operate on subsets of data. The stochastic version of the scaled conjugate gradient algorithm described in section 3.2.4 is a promising example of such an approach.

The second point might in the future turn out to be the best approach in general, because of the second-order convergence properties of these algorithms.

Chapter 4

Calculation of Hessian information

Second order derivatives of the error function appears frequently in many different aspects of neural networks. For instance, in second-order training algorithms like the scaled conjugate gradient [Møller 93a], in recent gradient descent acceleration techniques like the one described in section 3.1.7, in techniques for estimating generalization error [Moody 92], and in techniques for network pruning [Le Cun et al. 92], [Hassibi and Stork 93]. Finding methods for efficient extraction of Hessian information is therefore of profound importance. In this chapter, we present different techniques to calculate Hessian information. In this context, we present the results obtained in appendix C and D. Surprisingly, without too much calculation work it is possible to extract a lot of different information from the Hessian, such as exact multiplication by the Hessian, estimation of the smallest and largest eigenvalues and corresponding eigenvectors, and even estimation of the eigenvalue spectrum. The algorithm for multiplication by the Hessian, derived independently by Pearlmutter and Møller, is central in most of these extractions [Pearlmutter 93] [Møller 93c].

An exact formula for the calculation of the diagonal elements of the Hessian was derived in [Le Cun 89]. Later exact formulae for the calculation of all the Hessian elements were independently derived in [Bishop 92] and [Buntine and Weigend 91a]. We shortly summarize these results here. Let n_m^l and n_i^h be two units, where n_m^l is assumed to be in the same layer or in a lower layer than n_i^h. We want to calculate the second derivative $\frac{\partial^2 E_p}{\partial w_{ms}^{lr} \partial w_{ik}^{hj}}$ with respect to incoming weights w_{ms}^{lr} and w_{ik}^{hj}. The remaining terms of the Hessian, in which n_m^l is above n_i^h can be obtained from the symmetry of the Hessian without further calculation. By straightforward derivations using the chain rule, we get

$$\begin{aligned} \frac{\partial^2 E_p}{\partial w_{ms}^{lr} \partial w_{ik}^{hj}} &= \frac{\partial v_{pm}^l}{\partial w_{ms}^{lr}} \frac{\partial}{\partial v_{pm}^l} \left(\frac{\partial E_p}{\partial w_{ik}^{hj}} \right) \\ &= u_{ps}^r \left(\frac{\partial^2 E_p}{\partial v_{pi}^h \partial v_{pm}^l} u_{pk}^j + f'(v_{pk}^j) \frac{\partial E_p}{\partial v_i^h} \frac{\partial v_{pk}^j}{\partial v_{pm}^l} \right) \ . \end{aligned} \tag{4.1}$$

The following lemma describes how to calculate (4.1).

Lemma 6 *Assume that n_m^l is in the same layer or in a lower layer than n_i^h. The Hessian element $\frac{\partial^2 E_p}{\partial w_{ms}^{lr} \partial w_{ik}^{hj}}$ can be calculated by one forward and one backward propagation. The forward propagation is*

- $u^l_{pm} = f(v^l_{pm})$, *where* $v^l_{pm} = \sum_{n^r_s \in S^l_m} w^{lr}_{ms} u^r_{ps} + w^l_m$,
- $\alpha^{jl}_{km} = \sum_{n^r_s \in S^j_k} f'(v^r_{ps}) w^{jr}_{ks} \alpha^{rl}_{sm}$,

  $\alpha^{rr}_{ss} = 1$ *and* $\alpha^{rt}_{sv} = 0$, *for all* n^t_v *higher in the network than* n^r_s.

The backward propagation is

$$\frac{\partial^2 E_p}{\partial w^{lr}_{ms} \partial w^{hj}_{ik}} = u^r_{ps}(\delta^h_i f'(v^j_{pk}) \alpha^{jl}_{km} + u^j_{pk} \beta^{lh}_{mi}) ,$$

where δ^l_m *and* β^{lh}_{mi} *are*

- $\delta^l_m = f'(v^l_{pm}) \sum_{n^r_s \in T^l_m} w^{rl}_{sm} \delta^r_s$,

  $\delta^L_t = \frac{\partial E_p}{\partial v^L_{pt}}$, $\quad 1 \leq t \leq N_L$.

- $\beta^{lh}_{mi} = f''(v^l_{pm}) \alpha^{hl}_{im} \sum_{n^r_s \in T^l_m} w^{rl}_{sm} \delta^r_s + f'(v^l_{pm}) \sum_{n^r_s \in T^l_m} w^{rl}_{sm} \beta^{rh}_{si}$, $\quad l < L$,

  $\beta^{Lh}_{ti} = \alpha^{Lh}_{ti} \left(f''(v^L_{pt}) \frac{\partial E_p}{\partial u^L_t} + f'(v^L_{pt})^2 \frac{\partial^2 E_p}{(\partial u^L_t)^2} \right)$, $\quad 1 \leq t \leq N_L$.

Lemma 6 describes how to calculate the Hessian with respect to one pattern. The Hessian with respect to all patterns is of course a sum of these partial Hessian's. The lemma is constructed due to results in [Bishop 92]. For more detailed and general formulae see [Buntine and Weigend 91a]. The total number of forward and backward propagations scales with the number of units in the network. Each propagation cost $O(N)$ operations per pattern. The calculation of all the Hessian elements cost $O(N^2)$ operations per pattern.

Because of the time and memory requirements needed to calculate the Hessian, it is desirable to find methods that can extract Hessian information without explicitly having to calculate it. In many applications second order information is not needed in the form of the Hessian matrix itself, but rather in the form of the Hessian times a vector. Pearlmutter and Møller have independently shown that this product can be calculated exacly in $O(PN)$ time, without explicitly having to calculate the Hessian [Møller 93c], [Pearlmutter 93]. In the next section these results are summarized.

4.1 Hessian times a vector

The Hessian times a vector can be approximated by a one-sided difference equation of the form

$$E''(\mathbf{w}_k)\mathbf{d} \approx \frac{E'(\mathbf{w}_k + \sigma \mathbf{d}) \Leftrightarrow E'(\mathbf{w}_k)}{\sigma} \quad ,0 < \sigma \ll 1 \tag{4.2}$$

This approximation costs $O(PN)$ time to calculate and is in many cases a good estimate. The approximation can, however, be numericaly unstable even when high precision arithmetic is used. If the relative error of $E'(\mathbf{w}_k)$ is ε then the relative error of equation (4.2) can be as high as $\frac{2\varepsilon}{\sigma}$ [Ralston et al. 78]. So the relative error gets higher when σ is lowered. It is possible to calculate the Hessian times a vector exactly in the same order of time as the approximation. The following lemma states how.

Lemma 7 *The product* $E''(\mathbf{w}_k)\mathbf{d}$*, where* $\mathbf{d}$ *is a vector, can be calculated by one forward and one backward propagation. The forward propagation is*

$$u^l_{pm} = f(v^l_{pm}) \text{ , where } v^l_{pm} = \textstyle\sum_{n^r_s \in S^l_m} w^{lr}_{ms} u^r_{ps} + w^l_m,$$

$$\varphi^l_{pm} = \textstyle\sum_{n^r_s \in S^l_m} \left(d^{lr}_{ms} u^r_{ps} + w^{lr}_{ms} f'(v^r_{ps}) \varphi^r_{ps} \right) + d^l_m \quad , l > 0 \;\; , \qquad \varphi^0_{pi} = 0 \;\; , 1 \le i \le N_0 .$$

The backward propagation is

$$[E''(\mathbf{w}_k)\mathbf{d}]^{lh}_{mi} = \delta^l_{pm} f'(v^h_{pi}) \varphi^h_{pi} + (\mu^l_{pm} + \beta^l_{pm}) u^h_{pi} \; , \qquad [E''(\mathbf{w}_k)\mathbf{d}]^l_m = \mu^l_{pm} + \beta^l_{pm} \; ,$$

where μ^l_{pm} δ^l_{pm} *and* β^l_{pm} *are given by*

- $\mu^l_{pm} = f'(v^l_{pm}) \sum_{n^r_s \in T^l_m} w^{rl}_{sm} \mu^r_{ps} \quad , l < L$,

  $\mu^L_{pj} = \left(f'(v^L_{pj})^2 \frac{\partial^2 E_p}{(\partial u^L_{pj})^2} + f''(v^L_{pj}) \frac{\partial E_p}{\partial u^L_{pj}} \right) \varphi^L_{pj}$.

- $\delta^l_{pm} = f'(v^l_{pm}) \sum_{n^r_s \in T^l_m} w^{rl}_{sm} \delta^r_{ps} \quad , l < L$,

  $\delta^L_{pj} = \frac{\partial E_p}{\partial v^L_{pj}}$.

- $\beta^l_{pm} = \sum_{n^r_s \in T^l_m} \left(f'(v^l_{pm}) w^{rl}_{sm} \beta^r_{ps} + \left(d^{rl}_{sm} f'(v^l_{pm}) + w^{rl}_{sm} f''(v^l_{pm}) \varphi^l_{pm} \right) \delta^r_{ps} \right) \quad , l < L$,

  $\beta^L_{pj} = 0$.

A proof of the lemma is given in [Møller 93c]. The proof is a bit involved. Pearlmutter derives the results using a simpler technique, which we will describe here. Pearlmutter uses the fact that the Hessian times a vector can be written in the form

$$E''(\mathbf{w}_k)\mathbf{d} = \lim_{\sigma \rightarrow 0} \frac{E'(\mathbf{w}_k + \sigma\mathbf{d}) \Leftrightarrow E'(\mathbf{w}_k)}{\sigma} = \frac{\partial}{\partial\sigma} E'(\mathbf{w}_k + \sigma\mathbf{d}) \bigg|_{\sigma=0} \tag{4.3}$$

Let the linear differential operator $R\{\cdot\}$ be defined as

$$R\{f(\mathbf{w}_k)\} = \frac{\partial}{\partial\sigma} f(\mathbf{w}_k + \sigma\mathbf{d}) \bigg|_{\sigma=0} \tag{4.4}$$

We observe that this operator transforms the gradient of the error function into the desired product of the Hessian and the vector. Thus, an algorithm that calculates the gradient $E'(\mathbf{w}_k)$ can be transformed to an algorithm that calculates $E''(\mathbf{w}_k)\mathbf{d}$. The backpropagation algorithm stated in lemma 1 calculates the error gradient. Applying the $R\{\cdot\}$ operator to each formula in lemma 1 gives us exactly the result in lemma 7. Some useful rules to use when doing this is

$$\begin{aligned}
R\{f(c\mathbf{w}_k)\} &= cR\{f(\mathbf{w}_k)\} && (4.5)\\
R\{f(\mathbf{w}_k) + g(\mathbf{w}_k)\} &= R\{f(\mathbf{w}_k)\} + R\{g(\mathbf{w}_k)\} \\
R\{f(\mathbf{w}_k) g(\mathbf{w}_k)\} &= R\{f(\mathbf{w}_k)\} f(\mathbf{w}_k) + f(\mathbf{w}_k) R\{g(\mathbf{w}_k)\} \\
R\{f(g(\mathbf{w}_k))\} &= f'(g(\mathbf{w}_k)) R\{g(\mathbf{w}_k)\} \\
R\left\{\frac{df(\mathbf{w}_k)}{dt}\right\} &= \frac{dR\{f(\mathbf{w}_k)\}}{dt}
\end{aligned}$$

This elegant technique with the $R\{\cdot\}$ operator can be applied to other than feed-forward networks, such as recurrent networks and Boltzmann machines. It is, however, beyond the scope of this thesis to go into this discussion. We refer to [Pearlmutter 93] for such a discussion.

There are many applications for the algorithm stated in lemma 7. Just to mention a few, it can be used to improve line search algorithms, to calculate the learning rate in the scaled conjugate gradient algorithm (see section 3.2.3), to estimate eigenvalues and corresponding eigenvectors of the Hessian (see section 3.1.7), and even to estimate the whole eigenvalue spectrum. The spectrum can be estimated by means of an algorithm described in [Skilling 89]. It starts by calculating terms of the form $v_i = E^{''}(\mathbf{w}_k)^i v_0$, where v_0 is a random weight vector, and uses products $v_i \cdot v_j$ as estimates of the moments of the eigenvalue spectrum. Based on these estimations the spectrum can be recovered. Another application is adaptive preconditioning, which is described in appendix D and summarized in the next section.

4.1.1 Adaptive preconditioning

Both in gradient descent and conjugate gradient algorithms, there is a strong correlation between the condition number of the Hessian matrix and the convergence rate of the algorithms (see lemma 3 and lemma 5). The idea of preconditioning is to transform the Hessian into a matrix, which is well-conditioned, minimize the error in this transformed system and then at last transform back. The transformation is done on the Newtonian equations

$$E^{''}(\mathbf{w}_k)(\mathbf{w}_{k+1} \Leftrightarrow \mathbf{w}_k) = \Leftrightarrow E^{'}(\mathbf{w}_k) \ , \tag{4.6}$$

which originates, when minimizing the quadratic approximation to the error. The mostly used preconditioning scheme is *symmetric transformation.* The preconditioned system is then given by

$$A^T E^{''}(\mathbf{w}_k) A \mathbf{y} = \Leftrightarrow A^T E^{'}(\mathbf{w}_k) \ , \quad (\mathbf{w}_{k+1} \Leftrightarrow \mathbf{w}_k) = Ay \ , \tag{4.7}$$

where A is a $N \times N$ matrix, usually called the *preconditioning matrix.* A should be chosen such that $A^T E^{''}(\mathbf{w}) A$ has low condition number and is positive definite. Notice that symmetric preconditioning corresponds to minimizing the error in the direction of A times the original search direction. Several other preconditioning schemes exist, but no one seems to be preferable to others. Preconditioning has been well studied in the theory of conjugate gradient algorithms, see for example [Axelsson 80] and [Concus et al. 76]. Preconditioning can directly be build into the gradient descent and the conjugate gradient algorithms. Gradient descent with momentum combined with symmetric preconditioning is

$$\triangle \mathbf{w}_k = \Leftrightarrow \eta A A^T E^{'}(\mathbf{w}_k) + \alpha \triangle \mathbf{w}_{k-1} \ \ , \ \eta > 0 \ , \ \ 0 < \alpha < 1. \tag{4.8}$$

Conjugate gradient combined with the symmetric preconditioning is described in appendix D.

The problem with preconditioning is to determine an appropriate preconditioning matrix A. A should be chosen such that AA^T is close to the inverse Hessian. If $E^{''}(\mathbf{w})$ is positive definite, one could use $A^T = L^{-1}$, where L is the Choleski factor of $E^{''}(\mathbf{w})$ [Fletcher 75]. The Choleski factor costs, however, $O(N^3)$ time to compute, which is infeasible in a neural network context, and the Hessian is in many cases also indefinite.

A new idea, proposed in appendix D and [Møller 93d], is to adapt A during training. We will shortly describe this idea in the following.

First we have to choose the form of A, i.e., should A be symmetric, diagonal, tri-diagonal etc. It is clear, that A has to be sparse in some sense, since in a neural network context it is too costly to adapt a full $N \times N$ matrix. In appendix D, we chose A to be diagonal of the form $A = diag(\sigma(a_1), \sigma(a_2), \ldots, \sigma(a_N))$, where $\sigma(x)$ is a sigmoid function, but other forms could easily be considered.[1] Let the matrix G_k be defined as

$$G_k = A^T E^{''}(\mathbf{w}_k) A \ . \tag{4.9}$$

The adaptation of A should go in the direction of low condition number of G_k. Adapting A to minimize the condition number would require the estimation of the largest as well as the smallest eigenvalue of the Hessian. It is possible to estimate both terms with the Power method (see section 3.1.7), but the estimate of the smallest eigenvalue is often unstable. A better choice is to adapt A to minimize the function $M(A)$ defined by

$$M(A) = \left| \frac{\lambda_{max}}{<\lambda>} \right| \ , \tag{4.10}$$

where λ_{max} is the largest eigenvalue of G_k, $<\lambda>= \frac{1}{N} Tr(G_k)$ is the average eigenvalue and $Tr(G_k)$ is the trace of G_k. The gradient of $M(A)$ is

$$M^{'}(A) = \ sign(\frac{\lambda_{max}}{<\lambda>}) \ \frac{1}{<\lambda>^2} \left(\frac{d\lambda_{max}}{dA} <\lambda> \Leftrightarrow \frac{d<\lambda>}{dA} \lambda_{max} \right) . \tag{4.11}$$

The trace and the derivative of the trace of G_k are easy calculated by use of lemma 6. The estimate of the largest eigenvalue and its derivatives is the hardest part in this adaptation. The derivative of the largest eigenvalue with respect to one diagonal term a_i of A turns out to be

$$\frac{d\lambda_{max}}{da_i} = \frac{2\sigma'(a_i)[\mathbf{e}_{max}^T]_i [E^{''}(\mathbf{w}_k) A \mathbf{e}_{max}]_i}{|\mathbf{e}_{max}|^2} \ , \tag{4.12}$$

which can be calculated using lemma 7. The adaptation of A can be done by means of an *extended* Power method of the form

- Choose an initial random vector $\mathbf{e}_{max}^0$.
- Repeat the following steps for $t = 1, \ldots, T$, $T > 0$:
  - Repeat the following steps for $m = 1, \ldots, M$, $M > 0$:
    - $\mathbf{e}_{max}^m = G_k \mathbf{e}_{max}^{m-1}$,
    - $\lambda_{max}^m = \frac{(\mathbf{e}_{max}^{m-1})^T \mathbf{e}_{max}^m}{|\mathbf{e}_{max}^{m-1}|^2}$,
    - $\mathbf{e}_{max}^m = \frac{1}{\lambda_{max}^m} \mathbf{e}_{max}^m$,
  - $\frac{d\lambda_{max}^M}{dA} = \frac{(\mathbf{e}_{max}^M)^T \frac{dG_k}{dA} \mathbf{e}_{max}^M}{|\mathbf{e}_{max}^M|^2}$,
  - $a_i = a_i \Leftrightarrow \mu_{it} \ sign(\frac{\lambda_{max}^M}{<\lambda>}) \ \frac{1}{<\lambda>^2} \left(\frac{d\lambda_{max}^M}{da_i} <\lambda> \Leftrightarrow \frac{d<\lambda>}{da_i} \lambda_{max}^M \right)$, $\quad 1 \leq a_i \leq N$.

[1] This particular form of the diagonals assure that A is invertible and positive definite.

The inner loop estimates the largest eigenvalue, while the outer loop adapts A by gradient descent. Again the results in lemma 7 can be applied, this time to calculate the term $G_k \mathbf{e}_{max}^{m-1}$. The individual learning rate μ_{it} for each a_i is updated by

$$\mu_{it} = \begin{cases} 1.1 \; \mu_{it-1} & \text{if } \triangle a_i(t) \triangle a_i(t \Leftrightarrow 1) > 0 \\ 0.1 \; \mu_{it-1} & \text{otherwise} \end{cases} \tag{4.13}$$

As an additional constraint, the a_i's are limited to be in the range $[\Leftrightarrow 6, 6]$ in order to keep the derivatives $\sigma'(a_i)$ away from zero. In practice the process is run simultaneously with the updates of weights, so that A is updated say for every K weight updates. The extra time and memory requirements added to the learning algorithm per weight update is then $\frac{MT}{K} O(PN)$, which is in the same order as performing $\frac{MT}{K}$ gradient calculations more. The parameters T and M can often be set to small values. A configuration, that yields $\frac{MT}{K} = 1$ is not unusual.

Møller reports increase in convergence both for gradient descent and for the scaled conjugate gradient algorithm, when combined with adaptive preconditioning (see appendix D). The increase of convergence is most significant for gradient descent, where the speedup can be of several orders of magnitude. This increase can, however, not always be guaranteed. The process might even in some situations make the convergence worse. Problems arise, when the Hessian is indefinite, i.e., when $\lambda_{min} < 0$. Then there is no minimum to be found in the neighborhood of the current point, only a saddle point. It is not at all clear, how to interprete the condition number in this situation and how to predict the effect of a minimization of (4.10). Experiments in appendix D show that minimization of (4.10) often improves convergence, but not in general. The problem comes from intermediate eigenvalues which are near zero. If the minimization of (4.10) pushes these even closer to zero, then it is likely that the convergence will slow down instead of increase. A way to solve this problem might be to minimize the ratio of the largest eigenvalue to the eigenvalue closest to zero. So $M(A)$ could be defined as

$$M(A) = \left| \lambda_{max} \overline{\lambda}_{max} \right| \; , \tag{4.14}$$

where $\overline{\lambda}_{max}$ is the largest eigenvalue of the inverse of G_k. When the Hessian is positive definite this function is equal to the condition number of the Hessian. $\overline{\lambda}_{max}$ is, however, not easy to estimate in reasonable time. The power method could be used to estimate $\overline{\lambda}_{max}^2$ by applying it to the matrix $(\lambda_{max}^2 I \Leftrightarrow G_k^2)$. Unfortunately, this process takes too long to converge to be usable in practice. A practical technique to minimize the problem with indefinite Hessian matrices and convergence to saddle points is to restrict the preconditioning to flat regions in weight space. See appendix D for a further discussion about that.

The adaptive preconditioning method can also be combined with on-line update by exchanging each term $E''(\mathbf{w}_k)\mathbf{d}$, where $\mathbf{d}$ is a vector, with a running average as described in section 3.1.7. It should be possible to improve this method further by adapting A in a more efficient way, e.g., with second order methods, or by changing the definition of A to yield a stronger transformation than just a diagonal scaling. Another idea is to change the definition of the meta-error function $M(A)$ to a more sophisticated function. In section 3.2 under the analysis of the convergence of conjugate gradient algorithms, we observed that the more the eigenvalues were grouped, the faster convergence could be expected. Motivated by this observation, $M(A)$ could be defined as $var(\lambda)$, the variance of the eigenvalues. Clearly, minimization of the variance would get the eigenvalues closer around the mean. It is possible to estimate $var(\lambda)$ in an iterative fashion similar to the estimation of the largest eigenvalue. The following lemma due to Girard states how [Girard 89].

Lemma 8 *Let* $\mathbf{x}$ *denote a weight vector of* N *independent random values from the standard normal distribution.*[2] *Let* $T(\mathbf{x})$ *be defined by*

$$T(\mathbf{x}) = \frac{\mathbf{x}^T G_k \mathbf{x}}{\mathbf{x}^T \mathbf{x}} ,$$

Then $T(\mathbf{x})$ *is an unbiased estimator of* $<\lambda>$ *with variance* σ^2*:*

$$\sigma^2 = \frac{2}{N+2} var(\lambda).$$

So $M(A)$ could be defined as

$$M(A) = \frac{1}{K}\sum_{i=1}^{K} T(\mathbf{x}_i)^2 \Leftrightarrow \left(\frac{1}{K}\sum_{i=1}^{K} T(\mathbf{x}_i)\right)^2 . \qquad (4.15)$$

for some appropriate constant K. Minimization of $M(A)$ can again be done by means of an iterative method similar to the extended power method above, where the weights are updated simultanously in order to minimize the computational costs. This will be a subject for further research.

4.2 Inverse Hessian information

In many situations it is not the Hessian, but the inverse of the Hessian that is desirable to estimate. This is the case in most Newton inspired second-order optimization methods and in recent pruning techniques, such as *Optimal Brain Surgeon* (OBS) [Hassibi and Stork 93]. In the OBS pruning technique, the inverse Hessian is estimated based on prior knowledge about the error function. If the error function is the least mean square function, then the Hessian matrix can be approximated by its Jacobian in the form

$$E^{''}(\mathbf{w}_k) = \sum_{p=1}^{P} E_p^{''}(\mathbf{w}_k) \approx \sum_{p=1}^{P} E_p^{'}(\mathbf{w}_k) E_p^{'}(\mathbf{w}_k)^T . \qquad (4.16)$$

This corresponds to the well known Gauss-Newton approximation in nonlinear least squares optimization [Gill et al. 81]. The inverse can then be approximated iteratively by one pass through the training set by the following recursive formula[3]

$$H_{p+1}^{-1} = H_p^{-1} \Leftrightarrow \frac{H_p^{-1} E_{p+1}^{'}(\mathbf{w}_k) E_{p+1}^{'}(\mathbf{w}_k)^T H_p^{-1}}{1 + E_{p+1}^{'}(\mathbf{w}_k)^T H_p^{-1} E_{p+1}^{'}(\mathbf{w}_k)} , \qquad (4.17)$$

[2] See for example [Knuth 81] for algorithms to draw random variables from the normal distribution.

[3] We have here used the standard *Sherman-Morrison* inversion identity: $(A + \mathbf{a}\mathbf{b}^T)^{-1} = A^{-1} - \frac{A^{-1}\mathbf{a}\mathbf{b}^T A^{-1}}{1+\mathbf{b}^T A^{-1}\mathbf{a}}$.

where H_p^{-1} is the approximation of the inverse of the matrix $\sum_{i=1}^{p} E_i''(\mathbf{w}_k)$. It is then clear that H_P^{-1} is an approximation to $E''(\mathbf{w}_k)^{-1}$. Initially H_0 is picked as a "small" matrix which inverse is known. This approximation costs $O(PN^2)$ operations.

An alternative method, that is independent of the error function can be derived from Newton's method [Kreyszig 88]. To find the reciprocal x of a given number a, we may apply Newton's method on the function $f(x) = x^{-1} \Leftrightarrow a$. The iteration is then given by

$$x_{m+1} = x_m(2 \Leftrightarrow a x_m) \; .$$

This suggest an analogous iteration formula for determining the inverse $X = E''(\mathbf{w}_k)^{-1}$ of $E''(\mathbf{w}_k)$, namely

$$X_{m+1} = X_m(2I \Leftrightarrow E''(\mathbf{w}_k) X_m) \; . \tag{4.18}$$

This process converges if an only if X_0 is chosen such that the eigenvalues of $(I \Leftrightarrow E''(\mathbf{w}_k)X_m)$ are of absolute value less than 1. Unfortunately, a suitable choice of X_0 is generally difficult, and the method is mostly used for improving an inaccurate inverse obtained by another method.

4.2.1 Inverse Hessian times a vector

It is frequently necessary to find the inverse Hessian times a vector, which is the key calculation of Newton's method and also in the OBS technique. No exact formula exist for the calculation of $E''(\mathbf{w}_k)^{-1}\mathbf{d}$. It is not possible to find a formula like the one stated in lemma 7 for the calculation of the Hessian times a vector. Lemma 7 can, however, be used implicitly to estimate a solution iteratively by minimizing the function $||E''(\mathbf{w}_k)\mathbf{x} \Leftrightarrow \mathbf{d}||^2$ [Pearlmutter 93]. This can be done in at most N iterations, so all in all in order $O(PN^2)$ operations.

4.3 Conclusion

Hessian information is important in many aspects of neural network training. It can be used to improve convergence as well as the generalization ability of the network. It has often been assumed that it is too time consuming to extract Hessian information in a neural network context. This is not at all true. A lot of Hessian information can be extracted without explicitly having to calculate the matrix. This includes the Hessian times a vector, the largest and smallest eigenvalues of the Hessian and even the estimation of the whole eigenvalue spectrum.

Extraction of inverse Hessian information is a more complicated matter. It is possible to estimate the whole inverse matrix in less than $O(N^3)$, which is the order of time to calculate it exactly through numerical methods. The inverse Hessian times a vector can be estimated through an iterative method during training, but it is not clear how "safe" this method is.

Based on the above extraction methods, an adaptive preconditioning scheme was introduced in order to improve convergence. This approach is promising but needs to be improved and investigated further. One problem, that the scheme needs to consider more, is how to precondition indefinite Hessian matrices. One idea was to minimize the eigenvalues while keeping them away from zero. This involves the estimation of the largest eigenvalue of the inverse Hessian. Methods to do that exist, but they are not efficient. So trying to find an efficient method would be a natural way to continue this research.

The author feels that only "the top of the iceberg" has been investigated in this area and further research should be able to come up with even better methods to extract and use Hessian information.

Chapter 5

Different Error Functions

The choice of error function used in training feed-forward neural networks has a major influence on the convergence rate and on the final generalization ability of the network. This has been investigated by several researchers, such as [Solla et al. 88], [Yu and Simmons 90], [Brady and Raghavan 88], [Møller and Fahlman 93] and [Hampshire 92]. In this chapter, we summarize some of this work with focus on the results in [Møller and Fahlman 93] and appendix E.

Brady shows that gradient descent used on the least mean square error function can fail to separate families of vectors, even if there exists a hyperplane that separates them and no local minima exists. Brady's results are in fact not dependent on the training algorithm but only on the error function. The problem with the least mean square error function can be illustrated with an example where the optimal set of weights does not yield zero error. Then non-separating suboptimal solutions can exist with lower error than the optimal solution. So solutions with minimum least mean square error does not imply having a minimum number of misclassifications. The situation can be illustrated by the following simple figure [Hampshire and Waibel 90]. We have a network with two output units having output between 0 and 1. The outputs are mapped onto the x- and y-axis respectively. If the desired target pattern is (10) then all outputs to the right of the line $y = x$ can be considered correct. If and only if the contours of equal error are straight lines parallel to $y = x$, then there exist no regions with misclassification and lower error than other regions with correct classification. Hampshire defines error functions that satisfy such a condition to be *monotonic*. Hampshire strongly suggests that non-monotonic behavior in training can be the cause for the often seen "overlearning", i.e., where the recognition performance on a disjoint test set peaks and then degrades, while training set performance continues to improve.

The least mean square error function is non-monotonic as is the case with the entropy function described below. We shall later see that the exponential error function defined in [Møller and Fahlman 93] satisfies a *soft-monotonic* condition in the sense that the function is asymptotically monotonic in the limit for a certain parameter associated with the function.

Even if a solution with zero error exists, the problem with suboptimal solutions still exists in the form of local minima and in practice in the form of very flat regions in weight space. Suboptimal solutions in flat regions are often characterized by having a few patterns classified very wrong and many correct. The regions are flat because the network gradients are small for extreme wrong outputs. Minimization of the least mean square

Figure 5.1: Illustration of *non-monotonicity*. The x-axis is the output from the first unit and the y-axis is the output from the second. The curves corresponds to regions with equal least mean square error. Clearly, there are regions where the network misclassifies, but where the error is lower than in other regions where the network classifies correctly. For example, the error in the point x_1 is lower than the error in x_2. For a monotonic error function, the contours would have to be straight lines.

error function might very well converge to such regions because the training algorithms are *greedy* algorithms, updating weights in the direction of fastest error decrease, and no mechanism in the error function prevents the update of weights into these regions.

The entropy error function was introduced by Solla *et al.* to improve convergence in these flat regions [Solla et al. 88]. The entropy function for outputs between $\Leftrightarrow 1$ and 1 is given by

$$E(\mathbf{w}) = \frac{1}{2} \sum_{p=1}^{P} \sum_{j=1}^{J} \left((1 + t_{pj}) \log \frac{1 + o_{pj}}{1 + t_{pj}} + (1 \Leftrightarrow t_{pj}) \log \frac{1 \Leftrightarrow o_{pj}}{1 \Leftrightarrow t_{pj}} \right) \qquad (5.1)$$

The advantage with this function is, that the network gradients are cancelled out in the error gradients, so that error gradients for very wrong outputs are high. Notice that the entropy error function does not prevent the weights to converge into flat regions in weight space but makes it easier to escape from these regions.

5.1 The CFM error function

The classification figure of merit error function (CFM) applies to classification problems with an orthogonal output representation [Hampshire and Waibel 90].[1] This new approach to training neural networks, also often referred to as *differential learning*, maximizes a function of the minimum difference between the output from the unit representing the right class and other units output. In this way, learning focuses most on the reduction of misclassification, rather than on attempts to mimic the target outputs exactly. The CFM function can be written in the form

$$CFM(\mathbf{w}) = \sum_{p=1}^{P} \frac{\alpha}{1 + e^{-\beta \triangle_p + \zeta}} \ , \qquad (5.2)$$

where $\triangle_p$ is the minimum difference between correct output and other outputs, and α, β and ζ are constants. It can be shown that the CFM function is a monotonic error function. Hampshire provides experimental evidence that CFM minimizes the number of misclassifications better than the least mean square and the entropy function. Hampshire conjectures, that differential learning forms better and more general internal representations of the training data, yielding a better generalization ability of the network. However, no theoretical evidence for this exists. Training with the CFM error function is, unfortunately, much slower than training with the least mean square or entropy function.

5.2 The Exponential error function

The problem with the least mean square function and the entropy function is that nothing prevents the weights to converge to regions in weight space where a few of the patterns are misclassified in the extreme while the rest are classified correctly. Instead of insisting on strict monotonicity as in the CFM function, we can define error functions that satisfy a *soft-monotonic* condition, where a certain parameter controls the *degree* of monotonicity.

[1] An orthogonal output representation is one having exactly one output unit representing each single class. A classification is then considered correct if the output of the unit that represents the correct class is higher than all other outputs.

The key idea is to incorporate appropriate constraints into the error function, so that the weights are constrained away from bad regions in weight space. In appendix E and [Møller and Fahlman 93] an error function, called the exponential error function, was defined with these properties. The function is given by

$$E(\mathbf{w}) = \frac{1}{2}\sum_{p,j} e^{-\alpha(o_{pj}-t_{pj}+\beta)(t_{pj}+\beta-o_{pj})} \tag{5.3}$$

where α and β are positive parameters. The derivative to (5.3) with respect to a given o_{pj} is

$$\frac{dE(\mathbf{w})}{do_{pj}} = \Leftrightarrow\alpha(t_{pj} \Leftrightarrow o_{pj})e^{-\alpha(o_{pj}-t_{pj}+\beta)(t_{pj}+\beta-o_{pj})} \tag{5.4}$$

It is easy to see that the global minimum for (5.3) is when $t_{pj} = o_{pj}$, $\forall p, j$. β defines the width of the acceptable error around the desired target and α controls the steepness of the exponentially growing error in the penalized regions outside the interval. If α is small equation (5.4) resembles the derivative of the least square function. But the higher α gets the more active is the constraint imposed on the penalized regions. When no errors are in the penalized regions β is decreased, so that the outputs are pulled towards the targets. Note that the exponential error function indirectly balances the errors especially when α is large. A high α value gives large partial error derivatives inside the penalized regions and small partial error derivatives when outside the regions. So the higher the α value the more the errors will tend to arrange themselves around the boundary of the penalized regions. This gives a balanced distribution of the errors. For regression problems it is well known in statistics that a balanced set of errors can yield better generalization, this is often referred to as *variance heterogeneity* [Seber and Wild 89]. It is an open question whether this is true also for classification problems.

In the limit when α increases to infinity, the exponential error function is monotonic as illustrated in figure 5.2. Surely, for a fixed number of patterns in the training set, we can select a large enough α so that the error function is monotonic. The problem is how large α should be to ensure monotonicity in a given problem. Selecting too high a α slows down the convergence, because of too hard constraints imposed on the acceptable paths down to the minimum. On the other hand, too small a α results in non-monotonic behavior of the error function. One promising approach would be to adapt α similarly to the penalty parameters in constrained optimization, starting with a small α and then successively increasing α during training. This approach has not been tried yet. It seems that just setting α to a "reasonable" size yields good results.

In appendix E experimental evidence is provided showing that use of the exponential error function can increase convergence and improve the generalization ability of the networks. Notice that the exponential error function also works for non-classification problems and that the soft-monotonicity condition can be obtained for any accuracy required by adjustment of the β parameter.

5.3 Conclusion

It is well known that the least mean square error function and the entropy error function are both Bayes optimal in the sense that minimization with these functions produces

Figure 5.2: Illustration of *soft-monotonicity* in the limit for $\alpha \to \infty$. Notice that the condition for correct classification is different from that in figure 5.1.

solutions that approach the greatest lower bound on generalization error as the training set approaches infinity. This is of course a nice property but it is not necessarily relevant or at least not enough to classify an error function as being appropriate in practice. That an error function is Bayes optimal does not give any information about convergence properties, trajectories in weight space (e.g., if training often leads to local minima or flat regions in weight space), or generalization ability when trained on smaller sets of data. It assumes that the minimization of this error function always leads to nice regions in weight space, where the global minimum is to be found. This is not true for the least mean square neither for the entropy function, at least not on classification problems. Minimization of these functions often leads to suboptimal solutions characterized by having a few patterns classified extremely wrong and many correctly. The problem is that minimization of the error functions does not imply minimization of misclassifications. They are not *monotonic.*

We have described a monotonic error function, known as the CFM error function or differential learning originally invented by John Hampshire. This function works only on classification problems with orthogonal output representation. We have also described a new error function, called exponential error function, that exhibits a form of soft-monotonicity, where the monotonic behavior is dependent on the value of a certain parameter associated with the function.

Chapter 6

Conclusion

This introductory part of the Ph.D. thesis has explored the issue about efficient training of feed-forward neural networks. The results described in appendices A-E have been summarized in this context.

One main conclusion is, that *the* algorithm for training does not exist yet. It is still a problem dependent matter, what algorithm to use for training. But the choice of algorithm is, however, not that difficult anymore. One has to consider, if the training has to be in *on-line* or *off-line* mode. On-line is usually the best on classification problems characterized by training sets containing a lot of redundant information. If on-line mode is used, the choice is between the stochastic scaled conjugate gradient algorithm described in section 3.2.4 and appendix B and a carefully tuned on-line gradient descent algorithm combined with techniques like the reduction of large curvature components described in section 3.1.7. If off-line mode is used, then the choice is to use a second order algorithm, and then the scaled conjugate gradient algorithm described in section 3.2.3 is a good choice.

We have shown that second order information from the Hessian matrix in various ways can be extracted and used in training in an efficient way. Further research should definitely be done in this area. Efficient extraction of second order information can be used to speed up convergence and also to improve generalization. The adaptive preconditiong scheme described in section 4.1.1 is one example of such an approach to improve convergence. This method could be improved significantly by finding an efficient method for the estimation of eigenvalues of the inverse Hessian matrix.

The issues concerning use of different error functions should also be explored further. There is no doubt about the major impact, the choice of error function has on convergence and generalization. The CFM error function and the exponential error function are good starting points for research in this direction. Many questions need to be answered regarding error functions. How does the error distribution influence the quality of the solutions found ? What error distribution is best for classification problems and regressions problems ?

Appendix A

A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning

The paper [Møller 93a] was written in the fall of 1990, and was submitted to Neural Networks at that time. Because of an unfortunate mistake by the post office, the paper was first published in June 1993. The following is a slightly modified version of this paper.

A.1 Abstract

A supervised learning algorithm (Scaled Conjugate Gradient, SCG) is introduced. The performance of SCG is benchmarked against that of the standard backpropagation algorithm (BP) [Rumelhart et al. 86], the conjugate gradient algorithm with line search (CGL) [Johansson et al. 91] and the one-step Broyden-Fletcher- Goldfarb-Shanno memoryless quasi-Newton algorithm (BFGS) [Battiti 89]. SCG is fully automated, includes no user dependent parameters and avoids a time consuming line search, which CGL and BFGS use in each iteration in order to determine an appropriate step size. Experiments show that SCG is considerably faster than BP, CGL and BGFS.

A.2 Introduction

A.2.1 Motivation

Several adaptive learning algorithms for feed-forward neural networks have recently been discovered [Hinton 89]. Many of these algorithms are based on the gradient descent algorithm well known in optimization theory. They usually have a poor convergence rate and depend on parameters which have to be specified by the user, as no theoretical basis for choosing them exists. The values of these parameters are often crucial for the success of the algorithm. An example is the standard backpropagation algorithm [Rumelhart et al. 86] which often behaves very badly on large-scale problems and whose success depends of the user dependent parameters *learning rate* and *momentum constant*. The aim of this paper is to develop a supervised learning algorithm that eliminates some of these disadvantages.

From an optimization point of view, learning in a neural network is equivalent to minimizing a global error function, which is a multivariate function that depends on the weights in the network. This perspective gives some advantages in the development of effective learning algorithms because the problem of minimizing a function is well known in other fields of science, such as conventional numerical analysis [Watrous 87].

Since learning in realistic neural network applications often involves adjustment of several thousand weights, only optimization methods that are applicable to large-scale problems are relevant as alternative learning algorithms. The general opinion in the numerical analysis community is that especially one class of optimization methods, called the *Conjugate Gradient Methods*, are well suited to handle large-scale problems in an effective way [Hestenes and Stiefel 52], [Fletcher 75], [Gill et al. 81], [Powell 77].

Several conjugate gradient algorithms have recently been introduced as learning algorithms in neural networks [Johansson et al. 91], [Battiti 89], [Møller 90b]. Johansson, Dowla and Goodman describes the theory of general conjugate gradient methods and how to apply the methods in feed-forward neural networks. They conclude that the standard conjugate gradient method with line search (CGL) is an order of magnitude faster than the standard backpropagation algorithm (BP) [Rumelhart et al. 86] when tested on the parity problem. Battiti has introduced a variation of the standard conjugate gradient method, the one-step Broyden-Fletcher-Goldfarb-Shanno memoryless quasi-Newton algorithm (BFGS), as an alternative learning algorithm [Battiti 89]. He concludes that BFGS also yields a speed-up of about one order of magnitude compared to BP when tested on the parity problem. Both CGL and BFGS raise the calculation complexity per learning iteration considerably, since they have to perform a line search in order to determine an appropriate step size. A line search involves several calculations of either the global error function or its derivative, both of which raise the complexity.

This paper introduces a new variation of the conjugate gradient method (Scaled Conjugate Gradient, SCG), which avoids the line search per learning iteration by using a Levenberg-Marquardt approach [Fletcher 75] in order to scale the step size. During the development of SCG a tutorial to the theory of conjugate gradient related algorithms is given.

A.3 Optimization strategy

Most of the optimization methods used to minimize functions are based on the same strategy. The minimization is a local iterative process in which an approximation to the function in a neighbourhood of the current point in weight space is minimized. The approximation is often given by a first or second order Taylor expansion of the function. The idea of the strategy is illustrated in the pseudo algorithm presented below, which minimizes the error function $E(\mathbf{w})$ [Fletcher 75].

1. Choose initial weight vector $\mathbf{w}_1$ and set $k = 1$.

2. Determine a search direction $\mathbf{p}_k$ and a step size α_k so that $E(\mathbf{w}_k + \alpha_k \mathbf{p}_k) < E(\mathbf{w}_k)$.

3. Update vector: $\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \mathbf{p}_k$.

4. If $E'(\mathbf{w}_k) \neq 0$ then set $k = k + 1$ and go to 2
   else return $\mathbf{w}_{k+1}$ as the desired minimum.

Determining the next current point in this iterative process involves two independent steps. First a *search direction* has to be determined, i.e, in what direction in weight space do we want to go in the search for a new current point. Once the search direction has been found we have to decide how far to go in the specified search direction, i.e, a *step size* has to be determined.

A.4 The Backpropagation algorithm

If the search direction $\mathbf{p}_k$ in the above pseudo algorithm is set to the negative gradient $E^{'}(\mathbf{w}_k)$ and the step size α_k to a constant ϵ, then the algorithm becomes the gradient descent algorithm. In the context of neural networks this is the Backpropagation algorithm without momentum term. Minimization by gradient descent is based on the linear approximation $E(\mathbf{w}+\mathbf{y}) \approx E(\mathbf{w})+E^{'}(\mathbf{w})^T\mathbf{y}$, which is the main reason why the algorithm often shows poor convergence. Another reason is that the algorithm uses a constant step size, which in many cases is inefficient and makes the algorithm less robust. The inclusion of a momentum term in the backpropagation algorithm is an ad hoc attempt to force the algorithm to use second order information from the network. Unfortunately, the momentum term is not able to speed up the algorithm considerably, and causes the algorithm to be even less robust, because of the inclusion of another user dependent parameter, the momentum constant. Backpropagation including the momentum term will be referred to as BP.

Usually two versions of BP are considered, the ”off-line” version and the ”on-line” version. They differ in how often the weights are updated. The ”off-line” version updates the weights after all patterns have been propagated through the network, i.e, using information from all the patterns in the training set. The ”on-line” version updates after every single pattern, i.e., using only information from one pattern. The ”on-line” version is not consistent with optimization theory but it has nevertheless shown to be superior to the ”off-line” version on some specific problems. These problems seems to be characterized by big training sets containing a lot of redundant information [Le Cun 89]. The ”off-line” version is, however, superior on problems which does not have these properties.[1] This paper will use the ”off-line” version of BP in the comparison with other algorithms. For a more detailed discussion and comparison with the ”on-line” version of BP see [Møller 93b].

A.5 Conjugate direction methods

The Conjugate direction methods are also based on the above general optimization strategy but choose the search direction and the step size more carefully by using information from the second order approximation $E(\mathbf{w}+\mathbf{y}) \approx E(\mathbf{w})+E^{'}(\mathbf{w})^T\mathbf{y}+\frac{1}{2}\mathbf{y}^TE^{''}(\mathbf{w})\mathbf{y}$.

Quadratic functions have some advantages over general functions. Denote the quadratic approximation to E in a neighbourhood of a point $\mathbf{w}$ by $E_{qw}(\mathbf{y})$, so that $E_{qw}(\mathbf{y})$ is given by

$$E_{qw}(\mathbf{y}) = E(\mathbf{w})+E^{'}(\mathbf{w})^T\mathbf{y}+\frac{1}{2}\mathbf{y}^TE^{''}(\mathbf{w})\mathbf{y} \; . \tag{A.1}$$

[1] Such as the parity problem, which is used in this paper as a benchmark problem.

In order to determine minima to $E_{qw}(\mathbf{y})$, the critical points for $E_{qw}(\mathbf{y})$ must be found, i.e., the points where

$$E'_{qw}(\mathbf{y}) = E''(\mathbf{w})\mathbf{y} + E'(\mathbf{w}) = 0 \; . \tag{A.2}$$

The critical points are the solution to the linear system defined by (A.2). If a conjugate system is available the solution can be simplified considerably [Hestenes and Stiefel 52], [Johansson et al. 91]. Johansson, Dowla and Goodman show this in a very understandable manner. Let $\mathbf{p}_1, \ldots, \mathbf{p}_N$ be a conjugate system. Because $\mathbf{p}_1, \ldots, \mathbf{p}_N$ form a basis in $I\!R^N$, the step from a starting point $\mathbf{y}_1$ to a critical point $\mathbf{y}_*$ can be expressed as a linear combination of $\mathbf{p}_1, \ldots, \mathbf{p}_N$

$$\mathbf{y}_* \Leftrightarrow \mathbf{y}_1 = \sum_{i=1}^{N} \alpha_i \mathbf{p}_i \; , \quad \alpha_i \in I\!R. \tag{A.3}$$

Multiplying (A.3) with $\mathbf{p}_j^T E''(\mathbf{w})$ and substituting $E'(\mathbf{w})$ for $\Leftrightarrow E''''(\mathbf{w})\mathbf{y}_*$ gives

$$\begin{aligned} \mathbf{p}_j^T(\Leftrightarrow E'(\mathbf{w}) \Leftrightarrow E''(\mathbf{w})\mathbf{y}_1) &= \alpha_j \mathbf{p}_j^T E''(\mathbf{w})\mathbf{p}_j \quad \Rightarrow \\ \alpha_j &= \frac{\mathbf{p}_j^T(\Leftrightarrow E'(\mathbf{w}) \Leftrightarrow E''(\mathbf{w})\mathbf{y}_1)}{\mathbf{p}_j^T E''(\mathbf{w})\mathbf{p}_j} = \frac{\Leftrightarrow \mathbf{p}_j^T E'_{qw}(\mathbf{y}_1)}{\mathbf{p}_j^T E''(\mathbf{w})\mathbf{p}_j} \; . \end{aligned} \tag{A.4}$$

The critical point $\mathbf{y}_*$ can be determined in N iterative steps using (A.3) and (A.4). Unfortunately $\mathbf{y}_*$ is not necessarily a minimum, but can be a saddle point or a maximum. Only if the Hessian matrix $E''(\mathbf{w})$ is positive definite then $E_{qw}(\mathbf{y})$ has a unique global minimum. This can be realized by

$$\begin{aligned} E_{qw}(\mathbf{y}) &= E_{qw}(\mathbf{y}_* + (\mathbf{y} \Leftrightarrow \mathbf{y}_*)) \\ &= E(\mathbf{w}) + E'(\mathbf{w})^T(\mathbf{y}_* + (\mathbf{y} \Leftrightarrow \mathbf{y}_*)) \\ &\quad + \frac{1}{2}(\mathbf{y}_* + (\mathbf{y} \Leftrightarrow \mathbf{y}_*))^T E''(\mathbf{w})(\mathbf{y}_* + (\mathbf{y} \Leftrightarrow \mathbf{y}_*)) \\ &= E(\mathbf{w}) + E'(\mathbf{w})^T\mathbf{y}_* + E'(\mathbf{w})^T(\mathbf{y} \Leftrightarrow \mathbf{y}_*) + \frac{1}{2}\mathbf{y}_*^T E''(\mathbf{w})\mathbf{y}_* \\ &\quad + \frac{1}{2}\mathbf{y}_*^T E''(\mathbf{w})(\mathbf{y} \Leftrightarrow \mathbf{y}_*) + \frac{1}{2}(\mathbf{y} \Leftrightarrow \mathbf{y}_*)^T E''(\mathbf{w})\mathbf{y}_* + \frac{1}{2}(\mathbf{y} \Leftrightarrow \mathbf{y}_*)^T E''(\mathbf{w})(\mathbf{y} \Leftrightarrow \mathbf{y}_*) \\ &= E_{qw}(\mathbf{y}_*) + (\mathbf{y} \Leftrightarrow \mathbf{y}_*)^T(E''(\mathbf{w})\mathbf{y}_* + E'(\mathbf{w})) + \frac{1}{2}(\mathbf{y} \Leftrightarrow \mathbf{y}_*)^T E''(\mathbf{w})(\mathbf{y} \Leftrightarrow \mathbf{y}_*) \\ &= E_{qw}(\mathbf{y}_*) + \frac{1}{2}(\mathbf{y} \Leftrightarrow \mathbf{y}_*)^T E''(\mathbf{w})(\mathbf{y} \Leftrightarrow \mathbf{y}_*) \; , \end{aligned} \tag{A.5}$$

where the two last equalities come respectively from the fact that $E''(\mathbf{w})$ is symmetric and $E''(\mathbf{w})\mathbf{y}_* + E'(\mathbf{w}) = 0$ by (A.2). It follows from (A.5) that if $\mathbf{y}_*$ is a minimum then $\frac{1}{2}(\mathbf{y} \Leftrightarrow \mathbf{y}_*)^T E''(\mathbf{w})(\mathbf{y} \Leftrightarrow \mathbf{y}_*) > 0$ for every $\mathbf{y}$, hence $E''(\mathbf{w})$ has to be positive definite. The Hessian $E''(\mathbf{w})$ will in the following be assumed to be positive definite, if not otherwise stated.

The intermediate points $\mathbf{y}_{k+1} = \mathbf{y}_k + \alpha_k \mathbf{p}_k$ given by the iterative determination of $\mathbf{y}_*$ are in fact minima for $E_{qw}(\mathbf{y})$ restricted to every k-plane π_k: $\mathbf{y} = \mathbf{y}_1 + \alpha_1\mathbf{p}_1 + \cdots + \alpha_k\mathbf{p}_k$ [Hestenes and Stiefel 52]. How to determine these points recursively is shown in the following theorem. Its proof can be found in [Hestenes and Stiefel 52].

Theorem 1 *Let $\mathbf{p}_1,\ldots,\mathbf{p}_N$ be a conjugate system and $\mathbf{y}_1$ a point in weight space. Let the points $\mathbf{y}_2,\ldots,\mathbf{y}_{N+1}$ be recursively defined by*

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \alpha_k \mathbf{p}_k \ ,$$

where $\alpha_k = \frac{\mu_k}{\delta_k}$, $\mu_k = \Leftrightarrow \mathbf{p}_k^T E'_{qw}(\mathbf{y}_k)$ and $\delta_k = \mathbf{p}_k^T E''(\mathbf{w})\mathbf{p}_k$. Then $\mathbf{y}_{k+1}$ minimizes E_{qw} restricted to the k-plane π_k given by $\mathbf{y}_1$ and $\mathbf{p}_1,\ldots,\mathbf{p}_k$ [Hestenes and Stiefel 52].

The conjugate direction algorithm as proposed in [Hestenes and Stiefel 52] can be formulated as follows. Select an initial weight vector $\mathbf{y}_1$ and a conjugate system $\mathbf{p}_1,\ldots,\mathbf{p}_N$. Find successive minima for E_{qw} on the planes $\pi_1,\ldots,\pi_N$ using theorem 1, where π_k, $1 \leq k \leq N$, is given by $\mathbf{y} = \mathbf{y}_1 + \alpha_1\mathbf{p}_1 + \cdots + \alpha_k\mathbf{p}_k$, $\alpha_i \in I\!\!R$. The algorithm assures that the global minimum for a quadratic function is detected in, at most, N iterations. If all the eigenvalues of the Hessian $E''(\mathbf{w})$ fall into multiple groups with values of the same size, then there is a great probability that the algorithm terminates in much less than N iterations. Practice shows that this is often the case [Fletcher 75].

A.5.1 Conjugate gradients

The conjugate direction algorithm above assumes that a conjugate system is given. But how does one determine such a system? It is not necessary to know the conjugate weight vectors $\mathbf{p}_1,\ldots,\mathbf{p}_N$ in advance as they can be determined recursively. Initially $\mathbf{p}_1$ is set to the steepest descent vector $\Leftrightarrow E'_{qw}(\mathbf{y}_1)$. Then $\mathbf{p}_{k+1}$ is determined recursively as a linear combination of the current steepest descent vector $\Leftrightarrow E'_{qw}(\mathbf{y}_{k+1})$ and the previous direction $\mathbf{p}_k$. More precisely, $\mathbf{p}_{k+1}$ is chosen as the orthogonal projection of $\Leftrightarrow E'_{qw}(\mathbf{y}_{k+1})$ on the $(N \Leftrightarrow k)$-plane π_{N-k} conjugate to π_k. Theorem 2, given in [Hestenes and Stiefel 52], shows how this can be done.

Theorem 2 *Let $\mathbf{y}_1$ be a point in weight space and $\mathbf{p}_1$ and $\mathbf{r}_1$ equal to the steepest descent vector $\Leftrightarrow E'_{qw}(\mathbf{y}_1)$. Define $\mathbf{p}_{k+1}$ recursively by*

$$\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k \ ,$$

where $\mathbf{r}_{k+1} = \Leftrightarrow E'_{qw}(\mathbf{y}_{k+1})$, $\beta_k = \frac{|\mathbf{r}_{k+1}|^2 - \mathbf{r}_{k+1}^T\mathbf{r}_k}{\mathbf{r}_k^T\mathbf{r}_k}$ and $\mathbf{y}_{k+1}$ is the point generated in theorem 1. Then $\mathbf{p}_{k+1}$ is the steepest descent vector to E_{qw} restricted to the $(N \Leftrightarrow k)$-plane π_{N-k} conjugate to π_k given by $\mathbf{y}_1$ and $\mathbf{p}_1,\ldots,\mathbf{p}_k$ [Hestenes and Stiefel 52].

The conjugate vectors obtained using theorem 2 are often referred to as *conjugate gradient directions*. Combining theorem 1 and theorem 2 we get a *conjugate gradient algorithm*. In each iteration this algorithm can be applied to the quadratic approximation E_{qw} of the global error function E in the current point $\mathbf{w}$ in weight space. Because the error function $E(\mathbf{w})$ is non-quadratic the algorithm will not necessarily converge in N steps. If the algorithm has not converged after N steps, the algorithm is restarted, i.e., initializing $\mathbf{p}_{k+1}$ to the current steepest descent direction $\mathbf{r}_{k+1}$ [Hestenes and Stiefel 52], [Fletcher 75]. This also means that theorems 1-2 are only valid in the ideal case when the error E is equal to the quadratic approximation E_{qw}. This is, of course, not often the case but it does hold that the nearer the current point is to the minimum the better is the quadratic approximation E_{qw} of the error E. This property is in practice adequate to give a fast convergence. A standard conjugate gradient algorithm (CG) can now be described as follows.

1. Choose initial weight vector $\mathbf{w}_1$.
   Set $\mathbf{p}_1 = \mathbf{r}_1 = \Leftrightarrow E'(\mathbf{w}_1)$, $k = 1$.

2. Calculate second order information:

   $$\begin{aligned} \mathbf{s}_k &= E''(\mathbf{w}_k)\mathbf{p}_k, \\ \delta_k &= \mathbf{p}_k^T \mathbf{s}_k. \end{aligned}$$

3. Calculate step size:

   $$\begin{aligned} \mu_k &= \mathbf{p}_k^T \mathbf{r}_k, \\ \alpha_k &= \tfrac{\mu_k}{\delta_k}. \end{aligned}$$

4. Update weight vector:

   $$\begin{aligned} \mathbf{w}_{k+1} &= \mathbf{w}_k + \alpha_k \mathbf{p}_k, \\ \mathbf{r}_{k+1} &= \Leftrightarrow E'(\mathbf{w}_{k+1}). \end{aligned}$$

5. If k mod $N = 0$ then restart algorithm: $\mathbf{p}_{k+1} = \mathbf{r}_{k+1}$
   else create new conjugate direction:

   $$\begin{aligned} \beta_k &= \tfrac{|\mathbf{r}_{k+1}|^2 - \mathbf{r}_{k+1}^T \mathbf{r}_k}{|\mathbf{r}_k|^2}, \\ \mathbf{p}_{k+1} &= \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k. \end{aligned}$$

6. If the steepest descent direction $\mathbf{r}_{k+1} \neq \mathbf{0}$ then set $k = k + 1$ and go to 2
   else terminate and return $\mathbf{w}_{k+1}$ as the desired minimum.

Several other formulas for β_k can be derived [Hestenes and Stiefel 52], [Fletcher 75], [Gill et al. 81], but when the conjugate gradient methods are applied to non-quadratic functions the above formula, called the Hestenes-Stiefel formula, for β_k is considered superior. When the algorithm shows poor development the formula forces the algorithm to restart because of the following relation

$$\mathbf{r}_{k+1} \approx \mathbf{r}_k \quad \Rightarrow \quad \beta_k \approx 0 \quad \Rightarrow \quad \mathbf{p}_{k+1} \approx \mathbf{r}_{k+1} \ . \tag{A.6}$$

For each iteration in CG the Hessian matrix $E''(\mathbf{w}_k)$ has to be calculated and stored. It is not desirable to calculate the Hessian matrix explicitly, because of the calculation complexity and memory usage involved; actually calculating the Hessian would demand $O(N^2)$ memory usage and $O(PN^2)$ in calculation complexity. Usually this problem is solved by approximating the step size with a line search. Using the fact that $\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \mathbf{p}_k$ is a minimum for the k-plane $\mathbf{p}_1, \ldots, \mathbf{p}_k$, it is possible to show that

$$E'(\mathbf{w}_{k+1})^T \mathbf{p}_k = 0 \tag{A.7}$$

(A.7) shows that α_k is the solution to

$$\min_{\alpha} E(\mathbf{w}_k + \alpha \mathbf{p}_k) \tag{A.8}$$

So α_k is the minimum for E along the line $\mathbf{w}_k + \alpha \mathbf{p}_k$. α_k is in fact only an approximated solution to (A.8) since E is non-quadratic. The techniques for solving (A.8) are known as *line search techniques* [Gill et al. 81]. Appendix A.A gives a description of the line search algorithm used in this paper. All line search techniques include at least one user dependent parameter, which determines when the line search should terminate. The value of this parameter is often crucial for the success of the line search.

A.5.2 The CGL algorithm

The conjugate gradient algorithm (CG) shown above is often used combined with line search. That means the step size is approximated with a line search technique avoiding the calculation of the Hessian matrix. Johansson et al. has used this scheme using a cubic interpolation algorithm [Johansson et al. 91]. We use the conjugate gradient algorithm combined with the safeguarded quadratic univariate minimization mentioned in appendix A.A. This algorithm will be referred to as CGL.

A.5.3 The BFGS algorithm

Battiti has proposed another method from the optimization literature known as the one-step Broyden-Fletcher-Goldfarb-Shanno memory-less quasi- Newton method (BFGS) [Battiti 89]. The algorithm is also based on conjugate directions combined with line search. The direction is updated by the following rule

$$\mathbf{p}_k = S_k\mathbf{r}_k + A_k\mathbf{y}_k + S_kB_k\mathbf{q}_k \ , \tag{A.9}$$

where $\mathbf{r}_k = \Leftrightarrow E'(\mathbf{w}_k)$, $\mathbf{y}_k = \mathbf{w}_k \Leftrightarrow \mathbf{w}_{k-1}$ and $\mathbf{q}_k = E'(\mathbf{w}_k) \Leftrightarrow E'(\mathbf{w}_{k-1})$. The coefficients S_k, A_k and B_k are defined as

$$\begin{aligned} A_k &= (1 + S_k\frac{\mathbf{q}_k^T\mathbf{q}_k}{\mathbf{y}_k^T\mathbf{q}_k})B_k \Leftrightarrow S_k\frac{\mathbf{q}_k^T\mathbf{r}_k}{\mathbf{y}_k^T\mathbf{q}_k}, \\ B_k &= \frac{\mathbf{y}_k^T\mathbf{r}_k}{\mathbf{y}_k^T\mathbf{q}_k}, \quad S_k = \frac{\mathbf{y}_k^T\mathbf{q}_k}{\mathbf{q}_k^T\mathbf{q}_k} \end{aligned} \tag{A.10}$$

S_k, which is referred to as the scaling factor, is not strictly necessary [Luenberger 84]. Battiti has used $S_k = 1$ with good results. Again the safeguarded quadratic univariate minimization algorithm has been used in our experiments to estimate an appropriate step size.

A.6 The SCG algorithm

It is possible to use another approach in estimating the step size than the line search technique. The idea is to estimate the term $\mathbf{s}_k = E''(\mathbf{w}_k)\mathbf{p}_k$ in CG with a non-symmetric approximation of the form

$$\mathbf{s}_k = E''(\mathbf{w}_k)\mathbf{p}_k \approx \frac{E'(\mathbf{w}_k + \sigma_k\mathbf{p}_k) \Leftrightarrow E'(\mathbf{w}_k)}{\sigma_k} \ , \quad 0 < \sigma_k \ll 1 \ . \tag{A.11}$$

The approximation tends in the limit to the true value of $E''(\mathbf{w}_k)\mathbf{p}_k$. The calculation complexity and memory usage of $\mathbf{s}_k$ are respectively $O(PN)$ and $O(N)$. If this strategy is combined with the standard conjugate gradient approach (CG) we get an algorithm directly applicable to a feed-forward neural network. This slightly modified version of the original CG algorithm will also be referred to as CG.

The CG algorithm was tested on an appropriate test problem. It failed in almost every case and converged to a non-stationary point. Cause of this failure is that the algorithm only works for functions with positive definite Hessian matrices, and that the quadratic approximations on which the algorithm works can be very poor when the current point is far from the desired minimum [Gill et al. 81]. The Hessian matrix for the global error function E has shown to be indefinite in different areas of the weight space, which explains why CG fails in the attempt to minimize E.

We propose a new solution to this problem. The approach is new not only in the context of learning in feed-forward neural networks but also in the context of the underlying optimization theory which we have discussed so far. The idea is to combine the model-trust region approach, known from the Levenberg- Marquardt algorithm,[2] with the conjugate gradient approach. Let us introduce a scalar λ_k in CG, which is supposed to regulate the indefiniteness of $E''(\mathbf{w}_k)$. This is done by setting

$$\mathbf{s}_k = \frac{E'(\mathbf{w}_k + \sigma_k \mathbf{p}_k) \Leftrightarrow E'(\mathbf{w}_k)}{\sigma_k} + \lambda_k \mathbf{p}_k \ , \tag{A.12}$$

and adjusting λ_k in each iteration looking at the sign of δ_k, which directly reveals if $E''(\mathbf{w}_k)$ is not positive definite. If $\delta_k \leq 0$ then the Hessian is not positive definite and λ_k is raised and $\mathbf{s}_k$ is estimated again. If the new $\mathbf{s}_k$ is renamed as $\overline{\mathbf{s}}_k$ and the raised λ_k as $\overline{\lambda}_k$ then $\overline{\mathbf{s}}_k$ is

$$\overline{\mathbf{s}}_k = \mathbf{s}_k + (\overline{\lambda}_k \Leftrightarrow \lambda_k)\mathbf{p}_k \ . \tag{A.13}$$

Assume in a given iteration that $\delta_k \leq 0$. It is possible to determine how much λ_k should be raised in order to get $\delta_k > 0$. If the new δ_k is renamed as $\overline{\delta}_k$ then

$$\begin{aligned} \overline{\delta}_k &= \mathbf{p}_k^T \overline{\mathbf{s}}_k = \mathbf{p}_k^T(\mathbf{s}_k + (\overline{\lambda}_k \Leftrightarrow \lambda_k)\mathbf{p}_k) = \delta_k + (\overline{\lambda}_k \Leftrightarrow \lambda_k)|\mathbf{p}_k|^2 > 0 \quad \Rightarrow \\ \overline{\lambda}_k &> \lambda_k \Leftrightarrow \frac{\delta_k}{|\mathbf{p}_k|^2} \ . \end{aligned} \tag{A.14}$$

(A.14) implies that if λ_k is raised with more than $\Leftrightarrow\frac{\delta_k}{|\mathbf{p}_k|^2}$ then $\overline{\delta}_k > 0$. The question is: how much should $\overline{\lambda}_k$ be raised to get an optimal solution? This question can not yet be answered, but it is clear that $\overline{\lambda}_k$ in some way should depend on λ_k, δ_k and $|\mathbf{p}_k|^2$. A choice found to be reasonable is

$$\overline{\lambda}_k = 2(\lambda_k \Leftrightarrow \frac{\delta_k}{|\mathbf{p}_k|^2}) \ . \tag{A.15}$$

This leads to

$$\begin{aligned} \overline{\delta}_k &= \delta_k + (\overline{\lambda}_k \Leftrightarrow \lambda_k)|\mathbf{p}_k|^2 = \delta_k + (2\lambda_k \Leftrightarrow 2\frac{\delta_k}{|\mathbf{p}_k|^2} \Leftrightarrow \lambda_k)|\mathbf{p}_k|^2 \\ &= \Leftrightarrow \delta_k + \lambda_k |\mathbf{p}_k|^2 > 0 \ . \end{aligned} \tag{A.16}$$

The step size is given by

$$\alpha_k = \frac{\mu_k}{\delta_k} = \frac{\mu_k}{\mathbf{p}_k^T \mathbf{s}_k + \lambda_k |\mathbf{p}_k|^2} \ , \tag{A.17}$$

with $\mathbf{s}_k$ given by formula (A.11). The values of λ_k directly scale the step size in such a way that the bigger λ_k is the smaller the step size, which agrees well with our intuition of the function of λ_k.

The quadratic approximation E_{qw}, on which the algorithm works, may not always be a good approximation to $E(\mathbf{w})$ since λ_k scales the Hessian matrix in an artificial way. A

[2] The Levenberg-Marquardt algorithm is a variation of the standard Newton algorithm.

mechanism to raise and lower λ_k is needed which gives a good approximation, even when the Hessian is positive definite. Define

$$\Delta_k = \frac{E(\mathbf{w}_k) \Leftrightarrow E(\mathbf{w}_k + \alpha_k \mathbf{p}_k)}{E(\mathbf{w}_k) \Leftrightarrow E_{qw}(\alpha_k \mathbf{p}_k)} = \frac{2\delta_k \left(E(\mathbf{w}_k) \Leftrightarrow E(\mathbf{w}_k + \alpha_k \mathbf{p}_k)\right)}{\mu_k^2} . \tag{A.18}$$

Here Δ_k is a measure of how well $E_{qw}(\alpha_k \mathbf{p}_k)$ approximates $E(\mathbf{w}_k + \alpha_k \mathbf{p}_k)$ in the sense, that the closer Δ_k is to 1, the better is the approximation. λ_k is raised and lowered following the formula

if $\Delta_k > 0.75$ then $\lambda_k = \frac{1}{4}\lambda_k$

if $\Delta_k < 0.25$ then $\lambda_k = \lambda_k + \frac{\delta_k(1-\Delta_k)}{|\mathbf{p}_k|^2}$.

The formula for $\Delta_k < 0.25$ increases λ_k such that the new step size is equal to the minimum to a quadratic polynomial fitted to $E^{'}(\mathbf{w}_k)^T \mathbf{p}_k$, $E(\mathbf{w}_k)$ and $E(\mathbf{w}_k + \alpha_k \mathbf{p}_k)$ [Williams 91]. The SCG algorithm is as shown below.

1. Choose weight vector $\mathbf{w}_1$ and scalars $0 < \sigma \leq 10^{-4}$, $0 < \lambda_1 \leq 10^{-6}$, $\overline{\lambda}_1 = 0$.
   Set $\mathbf{p}_1 = \mathbf{r}_1 = \Leftrightarrow E^{'}(\mathbf{w}_1)$, $k = 1$ and success = true.

2. If success then calculate second order information:

   $\sigma_k = \frac{\sigma}{|\mathbf{p}_k|}$,

   $\mathbf{s}_k = \frac{E^{'}(\mathbf{w}_k + \sigma_k \mathbf{p}_k) - E^{'}(\mathbf{w}_k)}{\sigma_k}$,

   $\delta_k = \mathbf{p}_k^T \mathbf{s}_k$.

3. Scale δ_k: $\delta_k = \delta_k + (\lambda_k \Leftrightarrow \overline{\lambda}_k)|\mathbf{p}_k|^2$.

4. If $\delta_k \leq 0$ then make the Hessian matrix positive definite:

   $\overline{\lambda}_k = 2(\lambda_k \Leftrightarrow \frac{\delta_k}{|\mathbf{p}_k|^2})$,

   $\delta_k = \Leftrightarrow \delta_k + \lambda_k |\mathbf{p}_k|^2$, $\quad \lambda_k = \overline{\lambda}_k$.

5. Calculate step size:

   $\mu_k = \mathbf{p}_k^T \mathbf{r}_k$,

   $\alpha_k = \frac{\mu_k}{\delta_k}$.

6. Calculate the comparison parameter: $\Delta_k = \frac{2\delta_k \left(E(\mathbf{w}_k) - E(\mathbf{w}_k + \alpha_k \mathbf{p}_k)\right)}{\mu_k^2}$.

7. If $\Delta_k \geq 0$ then a successful reduction in error can be made:

   $\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \mathbf{p}_k$,

   $\mathbf{r}_{k+1} = \Leftrightarrow E^{'}(\mathbf{w}_{k+1})$,

   $\overline{\lambda}_k = 0$, success = true.

   If k mod $N = 0$ then restart algorrithm: $\mathbf{p}_{k+1} = \mathbf{r}_{k+1}$
   else create new conjugate direction:

$\beta_k = \frac{|\mathbf{r}_{k+1}|^2 - \mathbf{r}_{k+1}^T \mathbf{r}_k}{|\mathbf{r}_k|^2}$,
$\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$.
If $\Delta_k \geq 0.75$ then reduce the scale parameter: $\lambda_k = \frac{1}{4}\lambda_k$.

else a reduction in error is not possible: $\overline{\lambda}_k = \lambda_k$, success = false.

8. If $\Delta_k < 0.25$ then increase the scale parameter: $\lambda_k = \lambda_k + \frac{\delta_k(1-\Delta_k)}{|\mathbf{p}_k|^2}$.

9. If the steepest descent direction $\mathbf{r}_k \neq \mathbf{0}$ then set $k = k + 1$ and go to 2
else terminate and return $\mathbf{w}_{k+1}$ as the desired minimum.

The value of σ should be as small as possible taking the machine precision into account. When σ is kept small ($\leq 10^{-4}$), experiments indicate that the value of σ is not critical for the performance of SCG. Because of that, SCG seems not to include any user dependent parameters which values are crucial for the success of the algorithm. This is a major advantage compared to the line search based algorithms which include that kind of parameters.

For each iteration there is one call of $E(\mathbf{w})$ and two calls of $E^{'}(\mathbf{w})$, which gives a calculation complexity per iteration of $O(5PN)$. When the algorithm is implemented this complexity can be reduced to $O(4PN)$ because the calculation of $E(\mathbf{w})$ can be built into one of the calculations of $E^{'}(\mathbf{w})$. In comparison with BP, SCG involves twice as much calculation work per iteration since BP has a calculation complexity of $O(2PN)$ per iteration. The calculation complexity of CGL and BFGS is about $O(3\text{-}15PN)$ since the line search, on average, involves 3-15 calls of $E(\mathbf{w})$ or $E^{'}(\mathbf{w})$ per iteration [Gill et al. 81].

When λ_k is zero, SCG is equal to the standard conjugate gradient algorithm (CG) shown before. Figure A.1 illustrates SCG functioning on an appropriate test problem.[3] Graph A) shows the error development versus learning iteration. The error decreases monotonically towards zero, which is characteristic for SCG because an error increase is not allowed. At several iterations the error is constant for one or two iterations.[4] In these instances the Hessian matrix has not been positive definite and λ_k has been increased using equation (A.13). The development of λ_k is shown in graph B). λ_k is varying between 0 and 25 iterations and is 0 in the rest of the minimization. This reveals that $E^{''}(\mathbf{w})$ has not been positive definite in the beginning of the minimization. This is not surprising since the closer the current point is to the desired minimum the greater is the probability that $E^{''}(\mathbf{w})$ is positive definite. We observe that whenever equation (A.13) is used to increase λ_k a large reduction in error is achieved immediately afterwards.

A.7 Test results

A.7.1 Comparison metric

In order to compare the performance of the different algorithms some kind of comparison metric is needed. Obviously the number of iterations is not a valid metric considering that the calculation complexity per iteration is not the same for any of the algorithms. Forward or backward passing of all the patterns through the network costs in the order

[3] The test problem was the logistic map problem described in [Battiti and Masulli 90]

[4] Iteration 6, 13, 20 and 24.

Figure A.1: SCG functioning on the logistic map problem. A) Error curve. B) λ curve.

of $O(PN)$ which is an order of magnitude greater than any other calculation in a given iteration for any of the algorithms. For that reason it seems reasonable to define the comparison metric using the amount of forward and backward passings of patterns. Define a *complexity unit* (cu) to be one forward or backward passing of all patterns in the training set. Then calculating the error costs 1 cu while calculating the derivative costs 2 cu. The complexity unit will be used to compare the performance of the different algorithms.

A.7.2 The parity problem

The aim of this test was to compare the performance of SCG with BP, CGL and BFGS. The algorithms were tested on 3, 4, 5, 6, 7, 8, 9 bit parity problems using 20 different initial weight vectors.[5] Three layer neural network architectures were used for each problem.[6] A training set containing all possible input patterns was used, i.e., 2^n patterns. The comparison metric described above was used in comparing the performance of the algorithms. The algorithms were terminated when the average error was less than 10^{-4} or the number of iterations had exceeded an appropriate large number of iterations. BP was run with learning rate 0.2 on parity 3-6, 0.05 on parity 7 and 0.01 on parity 8-9. The lowering of the learning rate was done in order to get BP to converge. The momentum was set to 0.9 for all problems. The line search parameter η in CGL and BFGS was set to 0.25. The results are illustrated in table 1. We observe that SCG is 2-3 times faster than CGL and BFGS on all problems.

It would also be interesting to see how the learning time is scaled by SCG, CGL, BFGS and BP. According to Hinton the learning time for BP should be approximately $O(PN^2)$, i.e., the total number of function calls, each costing $O(PN)$ time, should be approximately $O(N)$. This depends, however, on the nature of the task [Hinton 89], [Tesauro 87]. Judd shows that in the worst case it is exponential [Judd 87]. Figure A.2 uses logarithmic plot to illustrate the number of complexity units versus the number of input units for each of

[5] Though only 10 different initial weight vectors was used for BP on parity 8 and 9 because of the large amount of cpu-time involved in these experiments.

[6] n-n-1 architectures where n is the number of bits.

Bits	BP av/std/fai	SCG av/std/fai/sp	CGL av/std/fai/sp	BFGS av/std/fai/sp
3	3475/1020/0	413/306/1/8.4	1232/1383/1/2.8	736/473/0/4.7
4	16427/10185/1	1727/725/2/9.5	3320/3147/1/4.9	3004/3458/0/5.5
5	9864/5651/2	2131/1494/1/4.6	3682/2029/0/2.7	3246/2387/3/3.0
6	28671/20727/6	2811/1548/2/10.2	5435/6036/1/5.3	5601/3021/2/5.1
7	48478/38293/4	3801/3593/1/12.9	9903/12545/1/4.9	9343/10902/2/5.2
8	134130/64572/2	6206/3077/1/21.6	12518/14012/2/10.7	11426/8575/4/11.7
9	189453/53535/4	8105/5879/0/23.4	25855/22094/3/7.3	25748/24165/0/7.4

Table A.1: Results from the parity problem. av = average number of cu's. std = standard deviation. fai = number of failures. sp = speed-up relative to BP.

the two network architectures. The curves are clearly all sublinear indicating that all four algorithms scale polynomial on this particular problem.

The BP curve and the increasing speed-ups in table A.1 indicate that the scaling of BP is worse than for the other algorithms. There seems to be no significant difference in scaling of SCG, CGL and BFGS.

A.7.3 SCG performance versus different values of σ

The aim of this test was to determine how crucial the value of the σ-parameter is to the performance of SCG. 12 different values for σ were used on the parity 5 problem using 20 different initial weight vectors. The average results are shown in figure A.3. We observe that the average performance of SCG is not significantly affected when the value of σ is small ($\leq 10^{-4}$). For $\sigma \leq 10^{-4}$ the number of failures was in the range 0-2 and the standard deviation was 330. When σ was less than 10^{-12} roundoff errors began to have an effect.[7]

A.8 Conclusion

An optimization approach was used to introduce a learning algorithm (SCG) which is more effective than the standard backpropagation (BP), standard conjugate gradient with line search (CGL) and the one-step Broyden-Fletcher- Goldfarb-Shanno memoryless quasi-Newton algorithm (BFGS). SCG does not contain any user dependent parameters which values are crucial for the success of SCG. By using a step size scaling mechanism, SCG avoids a time consuming line search per learning iteration, which makes the algorithm faster than other second-order algorithms recently proposed (CGL,BFGS).

Acknowledgement

I would like to thank Brian Mayoh (AAU), Kim Plunkett (AAU) and Ole Østerby (AAU) for many good discussions and advice. I am also grateful to Roberto Battiti (University of

[7] All the experiments were run on a SUN-4 machine.

Figure A.3: SCG on parity 5 with different σ-values.

Trento, Italy) and Eric Johansson (LLNC) who have been very helpful giving constructive comments and advice.

Appendix A.A. Line search

A well known line search technique is that of Successive polynomial approximation where the function $E(\mathbf{w})$ is approximated by a simple function $e(\mathbf{w})$, which agrees exactly with $E(\mathbf{w})$ in either function value or function value and derivatives at a certain number of points. $e(\mathbf{w})$ is normally chosen to be a quadratic or cubic polynomial depending on whether or not the derivatives of $E(\mathbf{w})$ are available or easily calculated. We will use a quadratic polynomial and thereby avoiding calculating the derivatives of the error function, which involves twice as many calculations than calculating the error. Define the function $f(x)$ as

$$f(x) = E(\mathbf{w}_k + x\mathbf{p}_k) \; . \tag{A.19}$$

Assume that the minimum for $f(x)$ is bracketed by $(u, f(u))$, $(v, f(v))$ and that a third point $(x, f(x))$ in between is known. The minimum α for the quadratic polynomial passing through the three points is given by $x + s/q$ where s,q is

$$\begin{aligned} s &= (v \Leftrightarrow x)^2(f(u) \Leftrightarrow f(x)) \Leftrightarrow (u \Leftrightarrow x)^2(f(v) \Leftrightarrow f(x)) \qquad \text{(A.20)} \\ q &= 2\,((u \Leftrightarrow x)(f(v) \Leftrightarrow f(x)) \Leftrightarrow (v \Leftrightarrow x)(f(u) \Leftrightarrow f(x))) \end{aligned}$$

Successive applications of (A.20) can be shown to be superlinearly convergent when some mild conditions of $E(\mathbf{w})$ are satisfied. The disadvantage of the line search techniques is obviously that each successive step involves several calculations of the error which is of the order of $O(PN)$ calculations. Even initializing the line search algorithm, i.e, bracketing the minimum, can cost several calculations of the error. Because of the calculation complexity involved in each step, the line search should terminate after a small amount of steps. The termination criteria used in this paper is [Gill et al. 81]

$$E(\mathbf{w}_k) \Leftrightarrow E(\mathbf{w}_k + \alpha_j \mathbf{p}_k) <= \eta \alpha_j E^{'}(\mathbf{w}_k)\mathbf{p}_k \; , \quad 0 < \eta \leq \frac{1}{2} \; , \tag{A.21}$$

where α_j is the quadratic minimum for the j'th iteration in the successive line search. Terminating the line search before the actual minimum is found is called inexact line search. When the function to be minimized is non-quadratic, like the error function, making an exact line search is not worth while because the direction of search is also only an approximation to the exact direction. A slightly extended version of the quadratic line search technique is used in the experiments called *safeguarded quadratic univariate minimization* [Gill and Murray 74].

Appendix B

Supervised Learning on Large Redundant Training Sets

The paper [Møller 93b] was written in the spring of 1992 and has been published recently in International Journal of Neural Systems. The following is a non modified version of this paper.

B.1 Abstract

Efficient supervised learning on large redundant training sets requires algorithms where the amount of computation involved in preparing each weight update is independent of the training set size. Off-line algorithms like the standard conjugate gradient algorithms do not have this property, while on-line algorithms like the stochastic backpropagation algorithm do. A new algorithm combining the good properties of off-line and on-line algorithms is introduced.

B.2 Motivation

In the last few years many new learning algorithms for feed-forward neural networks have been introduced. One major approach has been to transform classical optimization algorithms into learning algorithms. In particular the conjugate gradient algorithms have in various versions shown to be effective [Battiti 92], [Møller 93a], [Battiti 92], [Johansson et al. 91]. Most of the results reported about the performance of these algorithms have been based on simulations made on small scale problems (parity, encoder, etc.). These results do not necessarily scale up to large scale problems. Figure B.1 illustrates simulations on a small scale and a medium scale problem using two conjugate gradient related algorithms (SCG [Møller 93a], BFGS [Battiti 89]) and backpropagation with off-line and on-line update respectively (OffBP, OnBP) [Rumelhart et al. 86]. We observe that SCG and BFGS are the most efficient algorithms for the small scale problem (4bit encoder).[1] The picture changes for the medium scale problem (1000 word Nettalk [Sejnowski and Rosenberg 87]).[2] SCG is still much better than Off-BP, but On-BP is sud-

[1] Even if the computational costs per iteration is taken into account the picture in figure B.1 will not change significantly.

[2] In this simulation only SCG, On-BP and Off-BP were tested.

Figure B.1: A) 4-bit encoder problem. B) 1000 word Nettalk problem.

denly the most efficient of them all. This phenomena is due to the characteristics of the nettalk problem and other medium to large scale realistic problems. It is characterized by a large and very redundant training set. As will be described in the next section, redundancy in training sets is reflected as redundant computations in algorithms like BFGS, SCG and Off-BP, that update weights based on information from the whole training set. Such algorithms are referred to as *off-line* algorithms, while algorithms like OnBP that update weights independent of the training set size are referred to as *on-line* algorithms.

Conjugate gradient algorithms and other second order algorithms are all off-line algorithms in the sense that there is no obvious way they are able to perform on-line updating of weights. The reason is that these algorithms choose a near optimal step-size before each weight update. Taking a near optimal step in the direction of error reduction of one pattern could crudely violate the error reduction on the whole training set. So second order algorithms are not as they stand able to use the redundancy present in the training set. It is, however, possible to combine the second order properties and the benefits of redundancy by introducing appropriate modifications to these algorithms. Such modifications are described in this paper.

B.3 Redundancy

Practical neural network problems are usually characterized by large and very regular training sets. These training sets often contain redundant information. Consider an extreme example where the training set is composed of two copies of the same subset [Le Cun 89]. Accumulating the partial gradients over the whole training set will cause redundant computations to be performed. This idea can be generalised to training sets where no precise repetition of the same pattern exist but where some redundancy is present.

The relationship between redundancy of training sets and redundant computations can be characterized as follows. The redundancy of the training set is reflected in the neural network as redundant error gradients, which again is reflected in the off-line learning algorithm as redundant computations because the gradients are successively accumulated before a weight update. The redundancy of the training set is a constant, but the redundancy of the gradients varies with learning, which again means that the redundant computations vary with learning. Measuring the constant redundancy of the training set would give a first estimate of the average amount of redundant computations to be expected during learning. There are obviously several different ways to construct such a measure of redundancy of training sets, one is to use Shannon's information theory [Shannon and Warren 64]. Consider a classification problem having M different classification classes and N discrete input vectors of length L each attribute having V possible values.[3] The *Conditional Population Entropy* (CPE) is defined as

$$CPE = \Leftrightarrow \sum_{m=1}^{M} p(c_m) \sum_{l=1}^{L} \sum_{v=1}^{V} p(x_v^l|c_m) \log p(x_v^l|c_m) \ , \tag{B.1}$$

where $p(c_m)$ is the probability that an input vector belongs to the mth class and $p(x_v^l|c_m)$ is the probability that the lth attribute of an input vector x has value v given that x belongs to the mth class. CPE is the information value given information about the categories [Mingers 89]. The smaller CPE the bigger is the redundancy. We could then define redundancy (RE) as

$$RE = \frac{\log V \Leftrightarrow \frac{CPE}{L}}{\log V} \ , \tag{B.2}$$

where $\log V$ is the necessary number of bits needed to code one attribute if all values are equally likely and $\frac{CPE}{L}$ is the average number of bits needed to code one attribute. Examples of redundancy of training sets are

n-bit parity: $RE = 0$

1000 word Nettalk: $RE = 0.88$

We observe that the redundancy measure gives zero redundancy on the n-bit parity problem. This is what we would expect for an appropriate redundancy measure which is based on the training set alone, since changing only one bit in a parity input vector gives a new classification. A redundancy of zero indicates that no redundant computations are made if the whole set of 2^n patterns is used in training. It is, however, well known that the parity problem can up to a very good generalisation be learned by just using 2^{n-1} patterns. This is an example that illustrates that the number of redundant computations made by off-line algorithms are not totally determined by the redundancy in the training set but also by the internal network dynamics. The nettalk data, which is a highly regular training set, gives a redundancy of 88%. Taking this as an estimate of the average redundant computations we get a hint of why we observe the inefficiency of the off-line algorithms on this problem.

The above redundancy measure could also be applied to the error gradient vectors. Instead of discrete attribute values we then have continuous values, which makes the situation more complicated. A way to handle the case would be to use intervals instead of discrete attribute values. Measuring the redundancy of the gradient vectors would give a much better estimate of the redundant computations. Doing that in each iteration of the learning process would, however, be very time consuming.

[3] The continuous case could be handled using intervals instead of discrete attribute values.

Figure B.2: Performance of SCG and On-BP on randomly generated training sets with various degrees of redundancy.

In order to illustrate that there is a correlation between the value of the above redundancy measure and the efficiency of the different learning algorithms, training sets of various degrees of redundancy were generated. The test problems were of order 12-bit input and 3-bit output. The network used for the simulations was a 12-8-3 network. Figure B.2 illustrates the results with SCG and OnBP.

Not surprisingly, we observe that the smaller the redundancy the harder the problem. The redundancy has a positive effect on OnBP in the beginning of the minimization and this effect is more significant the higher the redundancy. In both simulations there is a turning point where from that point on SCG is more efficient. At the beginning of the minimization the error is usually far from the desired minimum which means: A) The direction of search does not have to be as accurate as when closer to the minimum which favors the on-line update. B) The second order properties of SCG do not have a significant effect when far away from the minimum.

B.4 Stochastic SCG method

In the design of an algorithm that combines the good properties of off-line and on-line algorithms we will use a natural restriction of the possible approaches. The computations involved in preparing a weight update have to be independent of the training set size. This automatically restricts us to stochastic, heuristic approaches because only a small part of the whole training set is available for us at one specific time. Off-line and on-line algorithms can be viewed as two extremes of a more general approach of learning; an approach which we will call *block update*. A block update with blocksize B is a weight update based on a block of B patterns drawn from the training set. On-line update corresponds to a block update with blocksize 1 and off-line update to a block update with blocksize N, where N is the number of patterns in the training set. Observe that block update on varying blocks of data of maybe different size is equivalent to altering the minimization function in each iteration. Based on experiments in physics, Jerome Karle has shown that such methods can extend the range of convergence for the least square minimization techniques in nonlinear systems [Karle 91]. Haffner *et al.* has applied a block update scheme combined with back-propagation on a large speech recognition problem with good results [Haffner et al. 88].

We want to find an appropriate blocksize B so that a block update would decrease the total error of the training set with great confidence. The blocksize B has to be big enough to ensure a safe weight update using a near optimal step size but small enough to avoid redundant computations. We would expect such a blocksize to be problem specific, i.e., the blocksize depends on the nature of the training set as well as the internal dynamics of the network. Because of that the blocksize is expected to vary with learning. Assume for a while that a procedure to determine such a blocksize is known. We then need to be able to combine the block update approach with a second order off-line algorithm. We here focus on the conjugate gradient methods, but other methods could be selected.

B.4.1 Conjugate Gradient with block update

Recall that the standard conjugate gradient method can be described as follows

1. Choose initial weight vector $\mathbf{w}_1$.
   Set $\mathbf{p}_1 = \mathbf{r}_1 = \Leftrightarrow E'(\mathbf{w}_1)$, $k = 1$.

2. Do line search:

   $$\alpha_k = \min_\alpha E(\mathbf{w}_k + \alpha \mathbf{p}_k).$$

3. Update weight vector:

   $$\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \mathbf{p}_k,$$
   $$\mathbf{r}_{k+1} = \Leftrightarrow E'(\mathbf{w}_{k+1}).$$

4. If $k \bmod N = 0$ then restart algorithm: $\mathbf{p}_{k+1} = \mathbf{r}_{k+1}$
   else create new conjugate direction:

   $$\beta_k = \frac{|\mathbf{r}_{k+1}|^2 - \mathbf{r}_{k+1}^T \mathbf{r}_k}{|\mathbf{r}_k|^2},$$
   $$\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k.$$

5. If the steepest descent direction $\mathbf{r}_k \neq \mathbf{0}$ then set $k = k + 1$ and go to 2
   else terminate and return $\mathbf{w}_{k+1}$ as the desired minimum.

Several different versions of this algorithm exist. They differ in how the step size α_k is determined. α_k is usually estimated by an inexact line search, which can be very time consuming. Møller has introduced another approach in estimating α_k which is based on a scaling mechanism [Møller 93a]. This algorithm, denoted Scaled Conjugate Gradient (SCG), will be used in what follows. We will shortly summarize this algorithm and refer to [Møller 93a] for a more detailed description. In SCG the step size α_k is determined using the following formula

$$\begin{aligned} \alpha_k &= \frac{\mathbf{p}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{s}_k + \lambda_k |\mathbf{p}_k|^2} \qquad \text{(B.3)} \\ \mathbf{s}_k &= \frac{E'(\mathbf{w}_k + \sigma_k \mathbf{p}_k) \Leftrightarrow E'(\mathbf{w}_k)}{\sigma_k} \ , \quad 0 < \sigma_k \ll 1 \ . \end{aligned}$$

$\mathbf{s}_k$ is a one-sided difference approximation to $E''(\mathbf{w}_k)\mathbf{p}_k$. An algorithm for the exact calculation of $E''(\mathbf{w}_k)\mathbf{p}_k$ has recently been proposed independently by Pearlmutter and Møller [Pearlmutter 93], [Møller 93c]. This algorithm involves the same order of calculations as the approximation. λ_k is a scaling parameter whose function is similar to the scaling parameter found in the Levenberg Marquardt algorithm [Fletcher 75]. λ_k is in each iteration raised or lowered according to the success of the error reduction in the particular iteration.

The standard conjugate gradient algorithm (and SCG) as shown above is in its simplest form an off-line algoithm since the calculation of β_k involves the current gradient as well as the last gradient. Using block update with different blocksizes for each iteration would cause the defintion of β_k to be meaningless. However, if the residual vectors $\mathbf{r}_k$ are normalized then β_k is an approximation of the true β_k and the algorithm works very well [Kuhn and Herzberg 90], [Haffner et al. 88]. This normalization is done by using a normalized error function on each block: $E_B = \frac{1}{B}E(\mathbf{w})$, where B is the number of patterns in the block (blocksize) and $E(\mathbf{w})$ is the total error of the current block of data. Observe that each E_B is an approximation to the total mean error of the whole training set, which means that the direction vectors produced in the algorithm will be approximations to the real direction vectors. Considering that the real direction vectors can not be expected to be exactly conjugate themselves, because the error function is usually non-quadratic, this seems to be reasonable. The robustness of the algorithm will depend on how good the estimated residual vectors $\mathbf{r}_k = \Leftrightarrow\frac{1}{B}E'(\mathbf{w}_k)$, based on the smaller block of data, are compared to the real residual vectors.

A first naive approach in designing an algorithm would be to apply the SCG algorithm with block update iteratively on different blocks. Since the update is not based on the whole training set, an update is not certain to make a decrease of the total error. The near optimal choice of step size α_k in each iteration requires a validation procedure of each block update in order to prevent too large oscillations. Taking a near optimal step in the direction of error reduction of a block of patterns could crudely violate the error reduction on the whole training set. For that reason we want to estimate the probability P that a block update will cause a reduction in total error. As we shall see in the next section this can be done by drawing a random sample from the training set. Such a random sample will from now on be denoted *sample block*. Similarly will the block of data on which the block update is based be called *update block*. The details of the update-validation scheme is described in the next section. The sample block is used to estimate an *update probability* P_i. The weights are then updated with this probability. The patterns in the update block are selected from the last sample block so that the errors of the selected patterns are uniformly distributed. Figure B.3 illustrates the idea of this scheme.

B.4.2 Update validation

Define μ_i as the total mean error in a given iteration i in the minimization process. Validation of a block update is equivalent to estimating the update probability

$$P_i = P(\mu_i < \mu_{i-1}) . \tag{B.4}$$

P_i can be estimated by drawing a simple random sample without replacement. Assume that the estimated mean error $\hat{\mu}_i$, i.e., the mean error of the sample, is normal distributed

Figure B.3: Update validation scheme.

around μ_i with standard error σ_i. The normal distribution is adequate in most practical situations. Assume also that the sample size n_i of the sample is chosen such that

$$P(\frac{|\hat{\mu}_i \Leftrightarrow \mu_i|}{\mu_i} \geq r) = \alpha \ , \tag{B.5}$$

where r is called the *relative error* and α is a small probability (see appendix B.C). σ_i can be estimated by

$$\sigma_i \approx \frac{s_i}{\sqrt{n_i}}\sqrt{1 \Leftrightarrow \frac{n_i}{N}} \ , \tag{B.6}$$

where s_i is the standard error of the sample and N is the total number of patterns [Cochran 77]. Given that $\hat{\mu}_i$ is normal distributed we have

$$P(\hat{\mu}_i < x) = \frac{1}{\sigma_i\sqrt{2\pi}}\int_{-\infty}^{x} e^{-\frac{(\hat{\mu}_i-\mu_i)^2}{2\sigma_i^2}} d\hat{\mu}_i = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\frac{x-\mu_i}{\sigma_i}} e^{-\frac{1}{2}t^2} dt \ . \tag{B.7}$$

We are interested in finding $\hat{P}_i = P(\mu_i < \hat{\mu}_{i-1})$ which is an approximation of the real update probability P_i given by (B.4).

$$\begin{aligned} \hat{P}_i &= P(\mu_i < \hat{\mu}_{i-1}) = 1 \Leftrightarrow P(\mu_i > \hat{\mu}_{i-1}) = 1 \Leftrightarrow P(\hat{\mu}_i < \mu_i \Leftrightarrow \triangle\hat{\mu}_i) \\ &= 1 \Leftrightarrow \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\frac{-\triangle\hat{\mu}_i}{\sigma_i}} e^{-\frac{1}{2}t^2} dt = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\frac{\triangle\hat{\mu}_i}{\sigma_i}} e^{-\frac{1}{2}t^2} dt \ , \end{aligned} \tag{B.8}$$

where $\triangle\hat{\mu}_i = \hat{\mu}_{i-1} \Leftrightarrow \hat{\mu}_i$. $\hat{P}_i$ is an estimate of P_i, but how close is $\hat{P}_i$ to P_i? Recall that the sample size n_i was chosen so that (B.5) was true. (B.5) implies that

$$\mu_{i-1}(1 \Leftrightarrow r) \leq \hat{\mu}_{i-1} \leq \mu_{i-1}(1+r) \quad \Rightarrow \quad \hat{\mu}_{i-1}(1 \Leftrightarrow \frac{r}{1+r}) \leq \mu_i \leq \hat{\mu}_{i-1}(1 + \frac{r}{1 \Leftrightarrow r}) \tag{B.9}$$

with probability $(1 \Leftrightarrow \alpha) \approx 1$. (B.9) gives

$$\begin{aligned} \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{L} e^{-\frac{1}{2}t^2} dt \leq \quad P_i \quad &\leq \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{U} e^{-\frac{1}{2}t^2} dt \quad \Rightarrow \\ \hat{P}_i \Leftrightarrow \frac{1}{\sqrt{2\pi}}\int_{L}^{\frac{\triangle\hat{\mu}_i}{\sigma_i}} e^{-\frac{1}{2}t^2} dt \leq \quad P_i \quad &\leq \hat{P}_i + \frac{1}{\sqrt{2\pi}}\int_{\frac{\triangle\hat{\mu}_i}{\sigma_i}}^{U} e^{-\frac{1}{2}t^2} dt \end{aligned} \tag{B.10}$$

where $L = \frac{\hat{\mu}_{i-1}(1-\frac{r}{1+r})-\hat{\mu}_i}{\sigma_i}$ and $U = \frac{\hat{\mu}_{i-1}(1+\frac{r}{1-r})-\hat{\mu}_i}{\sigma_i}$. So the exact update probability P_i is bounded by (B.10) with probability $(1 \Leftrightarrow \alpha)$. Based on the above analysis of the uncertainty of $\hat{P}_i$ and the fact that formula (B.10) is an unbiased estimate of P_i, we decided to use a slightly upward biased estimate for $\hat{P}_i$. We define $\hat{P}_i$ as

$$\hat{P}_i = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{M} e^{-\frac{1}{2}t^2} dt \ , \tag{B.11}$$

where $M = \frac{\hat{\mu}_{i-1}(1+\frac{r}{m(1-r)})-\hat{\mu}_i}{\sigma_i}$. The constant m is referred to as *bias term*. In order to reduce the uncertainty of $\hat{P}_i$ we also replace $\hat{\mu}_{i-1}$ with $\hat{\mu}_{i1}$, which is defined as the mean error of the same sample block as $\hat{\mu}_i$ but before an update. In order to have consistency in the notation $\hat{\mu}_i$ is then renamed to $\hat{\mu}_{i2}$. So $\hat{\mu}_{i1}$ and $\hat{\mu}_{i2}$ are the mean errors of the current sample block before and after an update. This convention does not have any influence on the above analysis. The value of $\hat{P}_i$ can be estimated with high accuracy using a rational approximation [Abramowitz 64].

B.4.3 Estimate of blocksize

We now turn to the problem of determining an appropriate update blocksize. In each iteration we would like to measure the goodness of a particular blocksize. According to the update-validation scheme in figure B.3 a sample block selected from the training set in iteration k is partly used as an update block in iteration $k+1$. So a block of data is used twice in the algorithm, first as a sample block and then as an update block. Using this observation we can define a *gain function* $G_k(B)$ that measures the error change of a particular block during its total life as a sample block and an update block. To measure the goodness of a particular blocksize we define

$$G_k(B) = \frac{n_{k-1}(\hat{\mu}_{(k-1)1} \Leftrightarrow \hat{\mu}_{(k-1)2}) + B(e_{k1} \Leftrightarrow e_{k2})}{(n_{k-1}+B)Be_{k1}} \ , \tag{B.12}$$

where e_{k1} and e_{k2} are the mean errors of the update block before and after the current iteration. Note that if an update is rejected then e_{k2} is equal to e_{k1}.

That G_k is an appropriate estimate of the goodness of a blocksize we see from the following observations. Update block k is an uniformly distributed subset of sample block $k \Leftrightarrow 1$ (see figure B.3). This means that $e_{k1} \approx \hat{\mu}_{(k-1)2}$. In the case where the sample size n_{k-1} is equal to the blocksize B the update block k is equal to sample block $k \Leftrightarrow 1$ and $e_{k1} = \hat{\mu}_{(k-1)2}$. Assume that $B = n_k$, then

$$\frac{n_{k-1}(\hat{\mu}_{(k-1)1} \Leftrightarrow \hat{\mu}_{(k-1)2}) + B(e_{k1} \Leftrightarrow e_{k2})}{(n_{k-1}+B)Be_{k1}} = \frac{\hat{\mu}_{(k-1)1} \Leftrightarrow e_{k2}}{(n_{k-1}+B)\hat{\mu}_{(k-1)2}} \ . \tag{B.13}$$

This term is positive if $\hat{\mu}_{(k-1)1} > e_{k2}$, i.e., if the mean error of sample block $k \Leftrightarrow 1$, calculated when the algorithm is finished working on the block, is smaller than the mean error, calculated when the algorithm first sampled the block. When the sample size is greater than the blocksize we get a weighted version of (B.13).

The blocksize which in each iteration returns the maximum gain should be the new current blocksize. Since neither G_k nor some of the derivatives of G_k can be calculated we will use a heuristic approach to find the maximum. We assume that the optimal blocksize

Figure B.4: Iterative estimate of blocksize.

will be constant or just slowly changing during long periods of the minimization. This allows us to average the gain over a small number of iterations and use this as a more precise measure of the average goodness of a particular blocksize. Let $\hat{G}_j$ be

$$\hat{G}_j = \frac{1}{C}\sum_{k=i}^{C} \frac{n_{k-1}(\hat{\mu}_{(k-1)1} \Leftrightarrow \hat{\mu}_{(k-1)2}) + B(e_{k1} \Leftrightarrow e_{k2})}{(n_{k-1}+B)Be_{k1}} \,, \tag{B.14}$$

where C is a small constant.

Assume that an initial blocksize is known. The blocksize is updated after each C iterations in the minimization process. The main heuristic update approach to maximize $\hat{G}_j$ is based on a standard binary search method. The blocksize is doubled until a maximum of the gain has been bounded.[4] Call the endpoints and the corresponding gain values of this boundinginterval B_a, G_a and B_b, G_b respectively. In each iteration the blocksize is set to $\frac{B_a+B_b}{2}$ and the boundinginterval is updated appropriately according to whether the gain-value is greater than or less than the last calculated gain-value, which is either G_a or G_b. Figure B.4 illustrates the situation in the first few steps of the algorithm.

The maximum found might, however, not be a maximum during the whole minimization because the weights change in each iteration. The method is for that reason reset when the boundinginterval has collapsed, i.e. when $B_a = B_b$, and the estimated gain is negative. When the method is restarted a search for a new boundinginterval starting from the current blocksize is initiated. The method is presented in appendix B.B.

The author has also explored simpler heuristic schemes for update of blocksize, e.g., a linear update scheme without use of gain measures. This scheme was of the form $B_{k+c} = B_k + b$, where b and c are pre-determined constants. So the blocksize was increased by b every c iterations. If b and c are carefully tuned then this simpler scheme can work as well as the binary search scheme. The problem is, however, the tuning of b and c which is very problem dependent. To make things even worse the optimal values for b and c may very well vary during the minimization. The binary search scheme has no crucial problem dependent parameters like b and c and works in general better.

[4] In fact only until the estimated gain is positive.

B.5 Complexity

The algorithm described in section B.4 which is based on the standard scaled conjugate gradient algorithm (SCG) will from now on be denoted *stochastic scaled conjugate gradient* (SSCG). See appendix B.A for a detailed description of SSCG. In this section we will estimate the calculation complexity per epoch of SSCG. In each iteration the standard SCG algorithm is applied to a block of patterns of size B. In [Møller 93a] it was shown that the calculation complexity per iteration of SCG is $O(6N|W|)$, where N is the total number of patterns and $|W|$ is the number of weights in the network. So without taking the validation scheme into account the calculation complexity per iteration of SSCG is $O(6 <B> |W|)$, where $<B>$ is the average update blocksize. The validation scheme costs $O(2 <n> |W|)$, where $<n>$ is the average sample size. If we separate the iterations where an update was rejected then the total calculation complexity per epoch of SSCG is

$$\frac{1}{<P>}\frac{N}{<B>}O(6 <B> |W|+2 <n> |W|) = \frac{1}{<P>}O(6N|W|+2 <n><B> N|W|) \ , \quad \text{(B.15)}$$

where $<P>$ is the average update probability. Empirical results indicates that the fraction $<n><B>$ is in the range 1-2 and $<P>$ is in the range 0.5-1. The on-line backpropagation algorithm has a calculation complexity of $O(3N|W|)$ which means that SSCG involves approximately 3-6 times as much calculation work per epoch as On-BP.

B.6 Experiments

The SSCG algorithm was tested on the randomly generated data mentioned in section 2 and on two medium to large scale problems. The benchmarking algorithms were On-BP and SCG.

B.6.1 Random generated training sets.

SSCG, SCG and OnBP were tested on the randomly generated training sets also used in section B.3. Figure B.5 illustrates the results. The curves for SSCG does more or less follow the On-BP curve until the turning point where On-BP begins to flatten out. SSCG continues to fall as SCG. SSCG does indeed use the redundancy in the beginning of the minimization and its second order properties in the last part as desired.

B.6.2 The nettalk problem

The nettalk problem was first described in [Sejnowski and Rosenberg 87]. The training set consists of 1000 different words and their corresponding pronunciations given by their phoneme representations. This gives a total number of 5438 patterns. The architecture of the network employs 203 input, 30 hidden and 26 output units giving a total number of 6926 weights. SSCG and On-BP were trained for 100 epochs using 10 different initial weight vectors. The values of the user defined parameters of SSCG were: relative error $(r) = 0.1$, bias term $(m) = 2$ and gain constant $(C) = 7$. The values of the user defined parameters of On-BP were: learning rate = 0.1 and momentum = 0.9. The average error curves are shown in figure B.6. We observe that SSCG converges faster than On-BP during the 100 epochs. In every one of the 10 runs SSCG was faster than On-BP.

Figure B.6: The average mean error and corresponding standard deviation for 10 runs with SSCG and On-BP on the nettalk problem.

Figure B.7: Blocksize.

When we weight this result with the time each algorithm approximately is using per epoch the picture changes slightly. On-BP converges faster than SSCG on the first few epochs but gets slowly overtaken by SSCG.

The standard deviation in each epoch is also illustrated in figure B.6. We observe very little and almost constant deviation of On-BP and a bigger but clearly decreasing deviation of SSCG. The decreasing trend of the standard deviation of SSCG indicate that the updates gets less and less stochastic which corresponds well with the observations to be made in figure B.7. Figure B.7 illustrates the development of the blocksize for a typical run of SSCG. First we observe that the blocksize is stable for long periods of the minimization and seems to converge quickly towards stable points. Secondly the blocksize is clearly increasing making the updates less and less stochastic as the error is getting closer to a minimum. If we continued the process we would expect the blocksize to be equal to the total number of patterns and the algorithm would finally be totally deterministic.

B.6.3 Currency exchange rate prediction

A data set of 4476 daily exchange rates for German mark (DM) with respect to US Dollar was used to train a feed-forward network to predict exhange rates. The net had 20 inputs for past daily returns, 10 hidden units and 1 output unit. The results are illustrated in figure 8. Since the training set is generated with a "sliding window" technique we would expect that redundancy would have a major impact on the training. Surprisingly On-BP does not converge faster than Off-BP (we tried several different values of learning rate), while SSCG is clearly superior. It is not obvious why On-BP is not able to use the redundancy.

Figure B.8: Currency exchange rate training.

B.7 Conclusion

The efficiency of supervised learning algorithms on small scale problems does not necessarily scale up to large scale problems. The redundancy of large training sets is reflected as redundant gradient vectors in the network. Accumulating these gradient vectors implies redundant computations. In order to avoid these redundant computations a learning algorithm has to be able to update weights independent of the size of the training set. A stochastic learning algorithm with this property has been proposed. The algorithm is denoted stochastic scaled conjugate gradient (SSCG) and is based on the scaled conjugate gradient algorithm (SCG) given in [Møller 93a]. The algorithm can, however, also be applied using standard conjugate gradient methods. Experimentally it is shown that SSCG converges faster than the on-line backpropagation algorithm (On-BP) on two medium scale problems. Further experiments need to be performed to be able to conclude anything definite about the convergence properties of SSCG compared to On-BP. We conjecture that stochastic second order algorithms like SSCG is superior to On-BP on large scale problems.

This paper has only described one particular update validation scheme. Other update validation schemes are possible, e.g., update on the same block of data until an update is rejected. In classification problems where the number of different classifications M is small it might be useful to let the update blocksize be proportional to M, so that each class in average would be represented equally in the update block [Haffner et al. 88]. A variety of different schemes need to be investigated in the field of stochastic optimization in feed-forward networks.

Appendix B.A. Stochastic SCG algorithm

Let E_u and E_v be the functions that calculate the mean errors of the current update block and the current sample block respectively.

```
1. Select small initial blocksize $B_1$;
   Select randomly without replacement an update block of size $B_1$;
   $\mu_{01} = 0$; $\mu_{02} = 0$; $\hat{G}_1 = 0$;
   Select a small integer $C > 0$;
   Choose weight vector $\mathbf{w}_1$ and scalars $0 < \sigma \leq 10^{-4}$, $0 < \lambda_1 \leq 10^{-6}$, $\overline{\lambda}_1 = 0$;
   $\mathbf{r}_1 = \Leftrightarrow E'_u(\mathbf{w}_1)$; $\mathbf{p}_1 = \mathbf{r}_1$;
   $k = 1$; $j = 1$; success = true;

2. If success then calculate second order information

      $\sigma_k = \frac{\sigma}{|\mathbf{p}_k|}$,
      $\mathbf{s}_k = \frac{E'_u(\mathbf{w}_k + \sigma_k \mathbf{p}_k) - E'_u(\mathbf{w}_k)}{\sigma_k}$,
      $\delta_k = \mathbf{p}_k^T \mathbf{s}_k$.

3. Scale $\delta_k$: $\delta_k = \delta_k + (\lambda_k \Leftrightarrow \overline{\lambda}_k)|\mathbf{p}_k|^2$.

4. If $\delta_k \leq 0$ then make the Hessian matrix positive definite:

      $\overline{\lambda}_k = 2(\lambda_k \Leftrightarrow \frac{\delta_k}{|\mathbf{p}_k|^2})$,
      $\delta_k = \Leftrightarrow \delta_k + \lambda_k |\mathbf{p}_k|^2$ ,   $\lambda_k = \overline{\lambda}_k$.

5. Calculate step size:

      $\mu_k = \mathbf{p}_k^T \mathbf{r}_k$,
      $\alpha_k = \frac{\mu_k}{\delta_k}$ .

6. $e_{k1} = E_u(\mathbf{w}_k)$; $e_{k2} = E_u(\mathbf{w}_k + \alpha_k \mathbf{p}_k)$;

7. Calculate the comparison parameter: $\Delta_k = \frac{2\delta_k(e_{k1} - e_{k2})}{\mu_k^2}$.

8. If $\Delta_k \geq 0$ then

      $\hat{G}_j = \hat{G}_j + \frac{n_{k-1}(\hat{\mu}_{(k-1)1} - \hat{\mu}_{(k-1)2}) + B(e_{k1} - e_{k2})}{(n_{k-1} + B) B e_{k1}}$;
      if ($k$ mod $C = 0$) then
         $\hat{G}_j = \frac{1}{C}\hat{G}_j$;
         Estimate new blocksize $B_{k+1}$;
         $\hat{G}_{j+1} = 0$; $j = j + 1$;
      Estimate new sample size $n_k \geq B_{k+1}$;
      Draw a random sample without replacement of size $n_k$;
      $\mu_{k1} = E_v(\mathbf{w}_k)$; $\mu_{k2} = E_v(\mathbf{w}_k + \alpha_k \mathbf{p}_k)$;
      Select new update block of size $B_{k+1}$ uniform distributed from current sample;
      Estimate update probability $\hat{P}_k$;
      Draw random number $0 \leq r \geq 1$;
      if ($r \leq \hat{P}_k$) then $\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k \mathbf{p}_k$;
      else $\mathbf{w}_{k+1} = \mathbf{w}_k$;
```

```
            $\mathbf{r}_{k+1} = \Leftrightarrow E'_u(\mathbf{w}_{k+1})$;
            $\overline{\lambda}_k = 0$, success = true.
            If $k$ mod $N = 0$ then restart algorrithm: $\mathbf{p}_{k+1} = \mathbf{r}_{k+1}$
            else create new conjugate direction:
                $\beta_k = \frac{|\mathbf{r}_{k+1}|^2 - \mathbf{r}_{k+1}^T \mathbf{r}_k}{|\mathbf{r}_k|^2}$,
                $\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$.
            If $\Delta_k \geq 0.75$ then reduce the scale parameter: $\lambda_k = \frac{1}{4}\lambda_k$.
        else $\overline{\lambda}_k = \lambda_k$; success = false;
    9.  If $\Delta_k < 0.25$ then increase the scale parameter: $\lambda_k = \lambda_k + \frac{\delta_k(1-\Delta_k)}{|\mathbf{p}_k|^2}$.
    10. If the steepest descent direction $\mathbf{r}_k \neq \mathbf{0}$ then set $k = k + 1$ and go to 2
        else terminate and return $\mathbf{w}_{k+1}$ as the desired minimum.
```

Appendix B.B. Blocksize estimation

```
1. choose initial blocksize $B_0$;
   $B_a = 0$; $G_a = 0$; $B_b = 0$;
   direction = up; $j = 0$;
2. calculate $\hat{G}_j$;
3. if (direction=up) then
       if ($\hat{G}_j > G_a$) and ($\hat{G}_j < 0$) then
           $B_a = B_j$; $G_a = \hat{G}_j$;
           $B_{j+1} = 2B_j$;
       else
           direction = down;
           checkleft = false;
           $B_b = B_j$; $G_b = \hat{G}_j$;
4. if (direction=down) then
       if checkleft then condition = ($\hat{G}_j \leq G_a$);
       else condition = ($\hat{G}_j \geq G_b$);
       if condition then
           checkleft = false;
           $B_b = B_j$; $G_b = \hat{G}_j$;
       else
           checkleft = true;
           $B_a = B_j$; $G_a = \hat{G}_j$;
```

```
        $B_{j+1} = \frac{B_a+B_b}{2}$;
5. if ($|B_b B_a| \leq 1$) and ($\hat{G}_j < 0$) then
        direction = up;
        $B_a = B_j$; $G_a = \hat{G}_j$;
        $B_{j+1} = 2B_j$;
6. $j = j + 1$; terminate or goto 2.
```

Appendix B.C. Estimate of sample size

We want to estimate a sample size n_i such that $P(\frac{|\hat{\mu}_i - \mu_i|}{\mu_i} \leq r) = \alpha$, where α is a small probability and r is the relative error. Since $\hat{\mu}_i$ is assumed to be normal distributed we have

$$\sigma_i = \sqrt{1 \Leftrightarrow \frac{n_i}{N}} \frac{S_i}{\sqrt{n_i}} \; , \tag{B.16}$$

where S_i is the standard error for the whole training set in iteration i. Let t be defined as

$$P(|\hat{\mu}_i \Leftrightarrow \mu_i| \geq t\sigma_i) = \alpha \; , \tag{B.17}$$

then

$$r\mu_i = t\sigma_i = t\sqrt{1 \Leftrightarrow \frac{n_i}{N}} \frac{S_i}{\sqrt{n_i}} \; . \tag{B.18}$$

Solving for n_i gives

$$n_i = \frac{\left(\frac{tS_i}{r\mu_i}\right)^2}{1 + \frac{1}{N}\left(\frac{tS_i}{r\mu_i}\right)^2} \; . \tag{B.19}$$

Since S_i and μ_i are not known, n_i can not be calculated using the above formula. An approximation is

$$n_i = \frac{\left(\frac{ts_i}{r\mu_i}\right)^2}{1 + \frac{1}{N}\left(\frac{ts_i}{r\mu_i}\right)^2} \; , \tag{B.20}$$

where s_i is the standard error of the current sample block. Since s_i is biased upwards, n_i will also be biased upwards. Because of that, we do not allow the estimated sample size to exceed a certain fraction of the total number of patterns N, say 5 %. As illustrated in figure B.3, the patterns in the update block are selected uniformly from the last sample block which implies that the sample size has to be greater than or equal to the size of the update block. The initial sample size n_0 is found by drawing a random sample of a predefined size, say 1-2 % of the total number of patterns, and then estimating n_0 using B.20.

Appendix C

Exact Calculation of the Product of the Hessian Matrix and a Vector in $O(N)$ Time

The paper [Møller 93c] was written in the spring of 1993 and has been submitted to Neural Computation and published as a technical report at DAIMI, Aarhus University. The following is a non modified version of this paper.

C.1 Abstract

Several methods for training feed-forward neural networks require second order information from the Hessian matrix of the error function. Although it is possible to calculate the Hessian matrix exactly it is often not desirable because of the computation and memory requirements involved. Some learning techniques does, however, only need the Hessian matrix times a vector. This paper presents a method to calculate the Hessian matrix times a vector in $O(PN)$ time, where P is the number of patterns in the training set and N is the number of variables in the network. This is in the same order as the calculation of the gradient to the error function. The usefulness of this algorithm is demonstrated by improvement of existing learning techniques.

C.2 Introduction

The second derivative information of the error function associated with feed-forward neural networks forms an $N \times N$ matrix, which is usually referred to as the Hessian matrix. Second derivative information is needed in several learning algorithms, e.g., in some conjugate gradient algorithms [Møller 93a], and in recent network pruning techniques [MacKay 91b], [Hassibi and Stork 93]. Several researchers have recently derived formulae for exact calculation of the elements in the Hessian matrix [Buntine and Weigend 91a], [Bishop 92]. In the general case exact calculation of the Hessian matrix needs $O(PN^2)$ time and $O(N^2)$ in memory requirements. For that reason it is often not worth while explicitly to calculate the Hessian matrix and approximations are often made as described in [Buntine and Weigend 91a]. The second order information is not always needed in the form of the Hessian matrix. This makes it possible to reduce the time- and memory requirements needed to obtain this information. The scaled conjugate gradient algorithm [Møller 93a] and a training algorithm recently proposed by Le Cun involving estimation of eigenvalues to the Hessian matrix [Le Cun et al. 93] are good examples of this. The second order information needed here is always in the form of the Hessian matrix times a vector. In both methods the product of the Hessian and the vector is usually approximated by a one sided difference equation. This is in many cases a good approximation but can, however, be numerical unstable even when high precision arithmetic is used.

It is possible to calculate the Hessian matrix times a vector exactly without explicitly having to calculate and store the Hessian matrix itself. Through straightforward analytic evaluations we give explicit formulae for the Hessian matrix times a vector. We prove these formulae and give an algorithm that calculates the product. This algorithm has O(N) time- and memory requirements which is of the same order as the calculation of the gradient to the error function. The algorithm is a generalized version of an algorithm outlined by Yoshida, which was derived by applying an automatic differentiation technique [Yoshida 91]. The automatic differentiation technique is an indirect method of obtaining derivative information and provides no analytic expressions of the derivatives [Dixon and Price 89]. Yoshida's algorithm is only valid for feed-forward networks with connections between adjacent layers. Our algorithm works for feed-forward networks with arbitrary connectivity.

The usefulness of the algorithm is demonstrated by discussing possible improvements of existing learning techniques. We here focus on improvements of the scaled conjugate gradient algorithm and on estimation of eigenvalues of the Hessian matrix.

C.3 Notation

The networks we consider are multilayered feed-forward neural networks with arbitrary connectivity. The network $\aleph$ consist of nodes n_m^l arranged in layers $l = 0, \ldots, L$. The number of nodes in a layer l is denoted N_l. In order to be able to handle the arbitrary connectivity we define for each node n_m^l a set of *source nodes* and a set of *target nodes*.

$$\begin{aligned} S_m^l &= \left\{ n_s^r \in \aleph | n_s^r \text{ connects to } n_m^l, \;\; r < l, \;\; 1 \le s \le N_r \right\} \\ T_m^l &= \left\{ n_s^r \in \aleph | n_m^l \text{ connects to } n_s^r, \;\; r > l, \;\; 1 \le s \le N_r \right\} \end{aligned} \tag{C.1}$$

The training set accociated with network $\aleph$ is

$$\left\{ (u_{ps}^0, s = 1, \ldots, N_0, \;\; t_{pj}, j = 1, \ldots, N_L), p = 1, \ldots, P \right\} \tag{C.2}$$

The output from a node n_m^l when a pattern p is propagated through the network is

$$u_{pm}^l = f(v_{pm}^l) \text{ , where } v_{pm}^l = \sum_{n_s^r \in S_m^l} w_{ms}^{lr} u_{ps}^r + w_m^l, \tag{C.3}$$

and w_{ms}^{lr} is the weight from node n_s^r to node n_m^l. w_m^l is the usual *bias* of node n_m^l. $f(v_{pm}^l)$ is an appropriate activation function, e.g., the sigmoid. The net-input v_{pm}^l is chosen to be the usual weighted linear summation of inputs. The calculations to be made could, however, easily be extended to other definitions of v_{pm}^l. Let an error function $E(\mathbf{w})$ be

$$E(\mathbf{w}) = \sum_{p=1}^{P} E_p(u_{p1}^L, \ldots, u_{pN_L}^L, t_{p1}, \ldots, t_{pN_L}) \text{ ,} \tag{C.4}$$

where $\mathbf{w}$ is a vector containing all weights and biases in the network, and E_p is some appropriate error measure associated with pattern p from the training set.

Based on the chain rule we define some basic recursive formulae to calculate first derivative information. These formulae are used frequently in the next section. Formulae based on backward propagation are

$$\begin{aligned} \frac{\partial v^h_{pi}}{\partial v^l_{pm}} &= \sum_{n^r_s \in T^l_m} \frac{\partial v^h_{pi}}{\partial v^r_{ps}} \frac{\partial v^r_{ps}}{\partial v^l_{pm}} = f'(v^l_{pm}) \sum_{n^r_s \in T^l_m} w^{rl}_{sm} \frac{\partial v^h_{pi}}{\partial v^r_{ps}} && \text{(C.5)} \\ \frac{\partial E_p}{\partial v^l_{pm}} &= \sum_{n^r_s \in T^l_m} \frac{\partial E_p}{\partial v^r_{ps}} \frac{\partial v^r_{ps}}{\partial v^l_{pm}} = f'(v^l_{pm}) \sum_{n^r_s \in T^l_m} w^{rl}_{sm} \frac{\partial E_p}{\partial v^r_{s}} && \text{(C.6)} \end{aligned}$$

C.4 Calculation of the Hessian times a vector

This section presents an exact algorithm to calculate the vector $H_p(\mathbf{w})\mathbf{d}$, where $H_p(\mathbf{w})$ is the Hessian matrix of the error measure E_p, and $\mathbf{d}$ is a vector. The coordinates in $\mathbf{d}$ are arranged in the same manner as the coordinates in the weight vector $\mathbf{w}$.

$$\begin{aligned} H_p(\mathbf{w})\mathbf{d} &= \frac{d}{d\mathbf{w}}(\mathbf{d}^T \frac{dE_p}{d\mathbf{w}}) = \frac{d}{d\mathbf{w}}(\mathbf{d}^T \sum_{j=1}^{N_L} \frac{\partial E_p}{\partial v^L_{pj}} \frac{dv^L_{pj}}{d\mathbf{w}}) && \text{(C.7)} \\ &= \sum_{j=1}^{N_L} \frac{\partial^2 E_p}{(\partial v^L_{pj})^2} (\mathbf{d}^T \frac{dv^L_{pj}}{d\mathbf{w}}) \frac{dv^L_{pj}}{d\mathbf{w}} + \frac{\partial E_p}{\partial v^L_{pj}} (\frac{d^2 v^L_{pj}}{d\mathbf{w}^2}\mathbf{d}) \\ &= \sum_{j=1}^{N_L} \left(f'(v^L_{pj})^2 \frac{\partial^2 E_p}{(\partial u^L_{pj})^2} + f''(v^L_{pj}) \frac{\partial E_p}{\partial u^L_{pj}} \right) (\mathbf{d}^T \frac{dv^L_{pj}}{d\mathbf{w}}) \frac{dv^L_{pj}}{d\mathbf{w}} + \frac{\partial E_p}{\partial v^L_{pj}} (\frac{d^2 v^L_{pj}}{d\mathbf{w}^2}\mathbf{d}), \end{aligned}$$

The first and second terms of equation (C.7) will from now on be referred to as the $\mathbf{A}$- and $\mathbf{B}$-vector respectively. So we have

$$\begin{aligned} \mathbf{A} &= \sum_{j=1}^{N_L} \left(f'(v^L_{pj})^2 \frac{\partial^2 E_p}{(\partial u^L_{pj})^2} + f''(v^L_{pj}) \frac{\partial E_p}{\partial u^L_{pj}} \right) (\mathbf{d}^T \frac{dv^L_{pj}}{d\mathbf{w}}) \frac{dv^L_{pj}}{d\mathbf{w}} \quad \text{and} && \text{(C.8)} \\ \mathbf{B} &= \sum_{j=1}^{N_L} \frac{\partial E_p}{\partial v^L_{pj}} (\frac{d^2 v^L_{pj}}{d\mathbf{w}^2}\mathbf{d}). \end{aligned}$$

We first concentrate on calculating the $\mathbf{A}$-vector.

Lemma 9 *Let φ^l_{pm} be defined as $\varphi^l_{pm} = \mathbf{d}^T \frac{dv^l_{pm}}{d\mathbf{w}}$. φ^l_{pm} can be calculated by forward propagation using the recursive formula*

$$\varphi^l_{pm} = \textstyle\sum_{n^r_s \in S^l_m} \left(d^{lr}_{ms} u^r_{ps} + w^{lr}_{ms} f'(v^r_{ps}) \varphi^r_{ps} \right) + d^l_m \quad , l > 0 \quad , \qquad \varphi^0_{pi} = 0 \quad , 1 \le i \le N_0.$$

Proof. For input nodes we have $\varphi^0_{pi} = 0$ as desired. Assume the lemma is true for all nodes in layers $k < l$.

$$\begin{aligned} \varphi^l_{pm} &= \mathbf{d}^T \frac{dv^l_{pm}}{d\mathbf{w}} = \mathbf{d}^T (\sum_{n^r_s \in S^l_m} \frac{d}{d\mathbf{w}} (w^{lr}_{ms} u^r_{ps}) + \frac{dw^l_m}{d\mathbf{w}}) \\ &= \sum_{n^r_s \in S^l_m} \left(w^{lr}_{ms} f'(v^r_{ps}) \mathbf{d}^T \frac{dv^r_{ps}}{d\mathbf{w}} + d^{lr}_{ms} u^r_{ps} \right) + d^l_m = \sum_{n^r_s \in S^l_m} \left(d^{lr}_{ms} u^r_{ps} + w^{lr}_{ms} f'(v^r_{ps}) \varphi^r_{ps} \right) + d^l_m \end{aligned}$$

□

Lemma 10 *Assume that the φ^l_{pm} factors have been calculated for all nodes in the network. The* **A***-vector can be calculated by backward propagation using the recursive formula*

$$\mathbf{A}^{lh}_{mi} = \mu^l_{pm} u^h_{pi}, \qquad \mathbf{A}^l_m = \mu^l_{pm} \ ,$$

where μ^l_{pm} *is*

$$\mu^l_{pm} = f'(v^l_{pm}) \textstyle\sum_{n^r_s \in T^l_m} w^{rl}_{sm} \mu^r_{ps} \quad ,l < L \ ,$$

$$\mu^L_{pj} = \Big(f'(v^L_{pj})^2 \tfrac{\partial^2 E_p}{(\partial u^L_{pj})^2} + f''(v^L_{pj}) \tfrac{\partial E_p}{\partial u^L_{pj}}\Big) \varphi^L_{pj} \quad ,1 \leq j \leq N_L.$$

Proof.

$$\mathbf{A}^{lh}_{mi} \ = \ \sum_{j=1}^{N_L} \mu^L_{pj} \frac{\partial v^L_{pm}}{\partial w^{lh}_{mi}} = \Big(\sum_{j=1}^{N_L} \mu^L_{pj} \frac{\partial v^L_{pj}}{\partial v^l_{pm}} \Big) u^h_{pi} \qquad \Rightarrow \qquad \mu^l_{pm} = \sum_{j=1}^{N_L} \mu^L_{pj} \frac{\partial v^L_{pj}}{\partial v^l_{pm}}$$

For the output layer we have $\mathbf{A}^{Lh}_{ji} = \mu^L_{pj} u^h_{pi}$ as desired. Assume that the lemma is true for all nodes in layers $k > l$.

$$\begin{aligned} \mu^l_{pm} \ &= \ \sum_{j=1}^{N_L} \mu^L_{pj} \frac{\partial v^L_{pj}}{\partial v^l_{pm}} = \sum_{j=1}^{N_L} \mu^L_{pj} f'(v^l_{pm}) \sum_{n^r_s \in T^l_m} w^{rl}_{sm} \frac{\partial v^L_{pj}}{\partial v^r_{ps}} \\ &= \ f'(v^l_{pm}) \sum_{n^r_s \in T^l_m} w^{rl}_{sm} \Big(\sum_{j=1}^{N_L} \mu^L_{pj} \frac{\partial v^L_{pj}}{\partial v^r_{ps}} \Big) = f'(v^l_{pm}) \sum_{n^r_s \in T^l_m} w^{rl}_{sm} \mu^r_{ps} \end{aligned}$$

□

The calculation of the **B**-vector is a bit more involved but is basicly constructed in the same manner.

Lemma 11 *Assume that the φ^l_{pm} factors have been calculated for all nodes in the network. The* **B***-vector can be calculated by backward propagation using the recursive formula*

$$\mathbf{B}^{lh}_{mi} = \delta^l_{pm} f'(v^h_{pi}) \varphi^h_{pi} + \beta^l_{pm} u^h_{pi} \ , \qquad \mathbf{B}^l_m = \beta^l_{pm}$$

where δ^l_{pm} *and* β^l_{pm} *are*

$$\delta^l_{pm} = f'(v^l_{pm}) \textstyle\sum_{n^r_s \in T^l_m} w^{rl}_{sm} \delta^r_{ps} \quad ,l < L \ , \qquad \delta^L_{pj} = \tfrac{\partial E_p}{\partial v^L_{pj}} \quad ,1 \leq j \leq N_L.$$

$$\beta^l_{pm} = \textstyle\sum_{n^r_s \in T^l_m} \Big(f'(v^l_{pm}) w^{rl}_{sm} \beta^r_{ps} + \big(d^{rl}_{sm} f'(v^l_{pm}) + w^{rl}_{sm} f''(v^l_{pm}) \varphi^l_{pm} \big) \delta^r_{ps} \Big) \quad ,l < L \ ,$$

$$\beta^L_{pj} = 0 \quad ,1 \leq j \leq N_L$$

Proof. Observe that the **B**-vector can be written in the form

$$\mathbf{B} = \sum_{j=1}^{N_L} \frac{\partial E_p}{\partial v^L_{pj}} \Big(\frac{d^2 v^L_{pj}}{d\mathbf{w}^2} \mathbf{d} \Big) = \sum_{j=1}^{N_k} \frac{\partial E_p}{\partial v^L_{pj}} \frac{d\varphi^L_{pj}}{d\mathbf{w}}.$$

Using the chain rule we can derive analytic experessions for δ^l_{pm} and β^l_{pm}.

$$\begin{aligned}
\mathbf{B}^{lh}_{mi} &= \sum_{j=1}^{N_L} \frac{\partial E_p}{\partial v^L_{pj}} \frac{\partial \varphi^L_{pj}}{\partial w^{lh}_{mi}} = \sum_{j=1}^{N_L} \frac{\partial E_p}{\partial v^L_{pj}} \Big(\frac{\partial \varphi^L_{pj}}{\partial \varphi^l_{pm}} \frac{\partial \varphi^l_{pm}}{\partial w^{lh}_{mi}} + \frac{\partial \varphi^L_{pj}}{\partial v^l_{pm}} \frac{\partial v^l_{pm}}{\partial w^{lh}_{mi}} \Big) \\
&= \sum_{j=1}^{N_L} \frac{\partial E_p}{\partial v^L_{pj}} \Big(\frac{\partial \varphi^L_{pj}}{\partial \varphi^l_{pm}} f'(v^h_{pi}) \varphi^h_{pi} + \frac{\partial \varphi^L_{pj}}{\partial v^l_{pm}} u^h_{pi} \Big)
\end{aligned}$$

So if the lemma is true δ^l_{pm} and β^l_{pm} are given by

$$\delta^l_{pm} = \sum_{j=1}^{N_L} \frac{\partial E_p}{\partial v^L_{pj}} \frac{\partial \varphi^L_{pj}}{\partial \varphi^l_{pm}} \ , \qquad \beta^l_{pm} = \sum_{j=1}^{N_L} \frac{\partial E_p}{\partial v^L_{pj}} \frac{\partial \varphi^L_{pj}}{\partial v^l_{pm}}$$

The rest of the proof is done in two steps. We look at the parts concerned with the β^l_{pm} and δ^l_{pm} factors separately. For all output nodes we have $\delta^L_{pj} = \frac{\partial E_p}{\partial v^L_{pj}}$ as desired. For non output nodes we have

$$\begin{aligned}
\delta^l_{pm} &= \sum_{j=1}^{N_L} \frac{\partial E_p}{\partial v^L_{pj}} \sum_{n^r_s \in T^l_m} \frac{\partial \varphi^L_{pj}}{\partial \varphi^r_{ps}} \frac{\partial \varphi^r_{ps}}{\partial \varphi^l_{pm}} = \sum_{j=1}^{N_L} \frac{\partial E_p}{\partial v^L_{pj}} f'(v^l_{pm}) \sum_{n^r_s \in T^l_m} w^{rl}_{sm} \frac{\partial \varphi^L_{pj}}{\partial \varphi^r_{ps}} \\
&= f'(v^l_{pm}) \sum_{n^r_s \in T^l_m} w^{rl}_{sm} \sum_{j=1}^{N_L} \frac{\partial E_p}{\partial v^L_{pj}} \frac{\partial \varphi^L_{pj}}{\partial \varphi^r_{ps}} = f'(v^l_{pm}) \sum_{n^r_s \in T^l_m} w^{rl}_{sm} \delta^r_{ps}
\end{aligned}$$

Similiarly is $\beta^L_{pj} = 0$ for all output nodes as desired. For non output nodes we have

$$\begin{aligned}
\beta^l_{pm} &= \sum_{j=1}^{N_L} \frac{\partial E_p}{\partial v^L_{pj}} \sum_{n^r_s \in T^l_m} \Big(\frac{\partial \varphi^L_{pj}}{\partial v^r_{ps}} \frac{\partial v^r_{ps}}{\partial v^l_{pm}} + \frac{\partial \varphi^L_{pj}}{\partial \varphi^r_{ps}} \frac{\partial \varphi^r_{ps}}{\partial v^l_{pm}} \Big) \\
&= \sum_{j=1}^{N_L} \frac{\partial E_p}{\partial v^L_{pj}} \sum_{n^r_s \in T^l_m} \Big(f'(v^l_{pm}) w^{rl}_{sm} \frac{\partial \varphi^L_{pj}}{\partial v^r_{ps}} + \big(d^{rl}_{sm} f'(v^l_{pm}) + w^{rl}_{sm} f''(v^l_{pm}) \varphi^l_{pm} \big) \frac{\partial \varphi^L_{pj}}{\partial \varphi^r_{ps}} \Big) \\
&= \sum_{n^r_s \in T^l_m} \Big(f'(v^l_{pm}) w^{rl}_{sm} \sum_{j=1}^{N_L} \frac{\partial E_p}{\partial v^L_{pj}} \frac{\partial \varphi^L_{pj}}{\partial v^r_{ps}} + \big(d^{rl}_{sm} f'(v^l_{pm}) + w^{rl}_{sm} f''(v^l_{pm}) \varphi^l_{pm} \big) \sum_{j=1}^{N_L} \frac{\partial E_p}{\partial v^L_{pj}} \frac{\partial \varphi^L_{pj}}{\partial \varphi^r_{ps}} \Big) \\
&= \sum_{n^r_s \in T^l_m} \Big(f'(v^l_{pm}) w^{rl}_{sm} \beta^r_{ps} + \big(d^{rl}_{sm} f'(v^l_{pm}) + w^{rl}_{sm} f''(v^l_{pm}) \varphi^l_{pm} \big) \delta^r_{ps} \Big)
\end{aligned}$$

The proof of the formula for $\mathbf{B}^l_m$ follows easily from the above derivations and is left to the reader. □

We are now ready to give an explicit formula for calculation of the Hessian matrix times a vector. Let $\mathbf{Hd}$ be the vector $H_p(\mathbf{w})\mathbf{d}$.

Corollary 1 *Assume that the φ^l_{pm} factors have been calculated for all nodes in the network. The vector* $\mathbf{Hd}$ *can be calculated by backward propagation using the following recursive formula*

$$\mathbf{Hd}^{lh}_{mi} = \delta^l_{pm} f'(v^h_{pi}) \varphi^h_{pi} + (\mu^l_{pm} + \beta^l_{pm}) u^h_{pi} \ , \qquad \mathbf{Hd}^l_m = \mu^l_{pm} + \beta^l_{pm} \ ,$$

where δ^l_{pm}, μ^l_{pm} and β^l_{pm} are given as shown in lemma 10 and lemma 11.

Proof. By combination of lemma 10 and lemma 11. □

If we view first derivatives like $\frac{\partial E_p}{\partial u^l_{pm}}$ and $\frac{\partial E_p}{\partial v^l_{pm}}$ as already available information, then the formula for $\mathbf{Hd}$ can reformulated into a formula based only on one recursive parameter. First we observe that δ^l_{pm} and β^l_{pm} can be written in the form

$$\begin{aligned}\delta^l_{pm} &= \frac{\partial E_p}{\partial v^l_{pm}} \\ \beta^l_{pm} &= f'(v^l_{pm}) \sum_{n^r_s \in T^l_m} \left(w^{rl}_{sm} \beta^r_{ps} + d^{rl}_{sm} \frac{\partial E_p}{\partial v^r_{ps}} \right) + f''(v^l_{pm}) \varphi^l_{pm} \frac{\partial E_p}{\partial u^l_{pm}}\end{aligned} \tag{C.9}$$

Corollary 2 *Assume that the φ^l_{pm} factors have been calculated for all nodes in the network. The vector* $\mathbf{Hd}$ *can be calculated by backward propagation using the following recursive formula*

$$\mathbf{Hd}^{lh}_{mi} = \frac{\partial E_p}{\partial v^l_{pm}} f'(v^h_{pi}) \varphi^h_{pi} + \gamma^l_{pm} u^h_{pi} \ , \quad \mathbf{Hd}^l_m = \gamma^l_{pm} \ ,$$

where γ^l_{pm} is

$$\gamma^l_{pm} = f'(v^l_{pm}) \sum_{n^r_s \in T^l_m} \left(w^{rl}_{sm} \gamma^r_{ps} + d^{rl}_{sm} \frac{\partial E_p}{\partial v^r_{ps}} \right) + f''(v^l_{pm}) \varphi^l_{pm} \frac{\partial E_p}{\partial u^l_{pm}}$$

$$\gamma^L_{pj} = \left(f'(v^L_{pj})^2 \frac{\partial^2 E_p}{(\partial u^L_{pj})^2} + f''(v^L_{pj}) \frac{\partial E_p}{\partial u^L_{pj}} \right) \varphi^L_{pj}$$

Proof. By corollary 1 and equation C.9. □

The formula in corollary 2 is a generalized version of the one that Yoshida derived for feed-forward networks with only connections between adjacent layers. An algorithm that calculates $\sum_{p=1}^P H_p(\mathbf{w})\mathbf{d}$ based on corollary 1 is given below. The algorithm also calculates the gradient vector $\mathbf{G} = \sum_{p=1}^P \frac{dE_p}{d\mathbf{w}}$.

1. Initialize.

   $\mathbf{Hd} = \mathbf{0}; \quad \mathbf{G} = \mathbf{0}$

   Repeat the following steps for $p = 1, \ldots, P$.

2. Forward propagation.

   For nodes $i = 1$ to N_0 do: $\varphi^0_{pi} = 0$.

   For layers $l = 1$ to L and nodes $m = 1$ to N_l do:

   $v^l_{pm} = \sum_{n^r_s \in S^l_m} w^{lr}_{ms} u^r_{ps} + w^l_m \ , \qquad u^l_{pm} = f(v^l_{pm}),$

   $\varphi^l_{pm} = \sum_{n^r_s \in S^l_m} \left(d^{lr}_{ms} u^r_{ps} + w^{lr}_{ms} f'(v^r_{ps}) \varphi^r_{ps} \right) + d^l_m.$

3. Output layer.

For nodes $j = 1$ to N_L do

$$\delta^L_{pj} = \frac{\partial E_p}{\partial v^L_{pj}} , \quad \beta^L_{pj} = 0 , \quad \mu^L_{pj} = \Big(f'(v^L_{pj})^2 \frac{\partial^2 E_p}{(\partial u^L_{pj})^2} + f''(v^L_{pj}) \frac{\partial E_p}{\partial u^L_{pj}}\Big)\varphi^L_{pj}.$$

For all nodes $n^r_s \in S^L_j$ do

$$\mathbf{Hd}^{Lr}_{js} = \mathbf{Hd}^{Lr}_{js} + \delta^L_{pj} f'(v^r_{ps})\varphi^r_{ps} + \mu^L_{pj} u^r_{ps} , \quad \mathbf{Hd}^L_j = \mathbf{Hd}^L_j + \mu^L_{pj} ,$$

$$\mathbf{G}^{Lr}_{js} = \mathbf{G}^{Lr}_{js} + \delta^L_{pj} u^r_{ps} , \quad \mathbf{G}^L_j = \mathbf{G}^L_j + \delta^L_{pj}.$$

4. Backward propagation.

For layers $l = L \Leftrightarrow 1$ downto 1 and nodes $m = 1$ to N_l do:

$$\mu^l_{pm} = f'(v^l_{pm}) \textstyle\sum_{n^r_s \in T^l_m} w^{rl}_{sm} \mu^r_{ps} , \quad \delta^l_{pm} = f'(v^l_{pm}) \textstyle\sum_{n^r_s \in T^l_m} w^{rl}_{sm} \delta^r_{ps},$$

$$\beta^l_{pm} = \textstyle\sum_{n^r_s \in T^l_m} \Big(f'(v^l_{pm}) w^{rl}_{sm} \beta^r_{ps} + \big(d^{rl}_{sm} f'(v^l_{pm}) + w^{rl}_{sm} f''(v^l_{pm})\varphi^l_{pm}\big)\delta^r_{ps}\Big).$$

For all nodes $n^r_s \in S^l_m$ do

$$\mathbf{Hd}^{lr}_{ms} = \mathbf{Hd}^{lr}_{ms} + \delta^l_{pm} f'(v^r_{ps})\varphi^r_{ps} + (\mu^l_{pm} + \beta^l_{pm}) u^r_{ps} , \quad \mathbf{Hd}^l_m = \mathbf{Hd}^l_m + \mu^l_{pm} + \beta^l_{pm} ,$$

$$\mathbf{G}^{lr}_{ms} = \mathbf{G}^{lr}_{ms} + \delta^l_{pm} u^r_{ps} , \quad \mathbf{G}^l_m = \mathbf{G}^l_m + \delta^l_{pm}.$$

Clearly this algorithm has $O(PN)$ time- and memory requirements. More precisely the time complexity is about 2.5 times the time complexity of a gradient calculation alone.

C.5 Improvement of existing learning techniques

In this section we justify the importance of the exact calculation of the Hessian times a vector, by showing some possible improvements on two different learning algorithms.

C.5.1 The scaled conjugate gradient algorithm

The scaled conjugate gradient algorithm is a variation of a standard conjugate gradient algorithm. The conjugate gradient algorithms produce non-interfering directions of search if the error function is assumed to be quadratic. Minimization in one direction $\mathbf{d}_t$ followed by minimization in another direction $\mathbf{d}_{t+1}$ imply that the quadratic approximation to the error has been minimized over the whole subspace spanned by $\mathbf{d}_t$ and $\mathbf{d}_{t+1}$. The search directions are given by

$$\mathbf{d}_{t+1} = \Leftrightarrow E'(\mathbf{w}_{t+1}) + \beta_t \mathbf{d}_t , \tag{C.10}$$

where $\mathbf{w}_t$ is a vector containing all weight values at time step t and β_t is

$$\beta_t = \frac{|E'(\mathbf{w}_{t+1})|^2 \Leftrightarrow E'(\mathbf{w}_{t+1})^T E'(\mathbf{w}_t)}{|E'(\mathbf{w}_t)|^2} \tag{C.11}$$

In the standard conjugate gradient algorithms the step size ϵ_t is found by a line search which can be very time consuming because this involves several calculations of the error and or the first derivative. In the scaled conjugate gradient algorithm the step size is estimated by a scaling mechanism thus avoiding the time consuming line search. The step size is given by

$$\epsilon_t = \frac{\Leftrightarrow \mathbf{d}_t^T E'(\mathbf{w}_t)}{\mathbf{d}_t^T \mathbf{s}_t + \lambda_t |\mathbf{d}_t|^2} \ , \tag{C.12}$$

where $\mathbf{s}_t$ is

$$\mathbf{s}_t = E''(\mathbf{w}_t)\mathbf{d}_t. \tag{C.13}$$

ϵ_t is the step size that minimizes the second order approximation to the error function. λ_t is a scaling parameter whose function is similar to the scaling parameter found in Levenberg-Marquardt methods [Fletcher 75]. λ_t is in each iteration raised or lowered according to how good the second order approximation is to the real error. The weight update formula is given by

$$\triangle \mathbf{w}_t = \epsilon_t \mathbf{d}_t \tag{C.14}$$

$\mathbf{s}_t$ has up til now been approximated by a one sided difference equation of the form

$$\mathbf{s}_t = \frac{E'(\mathbf{w}_t + \sigma_t \mathbf{d}_t) \Leftrightarrow E'(\mathbf{w}_t)}{\sigma_t} \quad ,0 < \sigma_t \ll 1 \tag{C.15}$$

$\mathbf{s}_t$ can now be calculated exactly by applying the algorithm from the last section. We tested the SCG algorithm on several test problems using both exact and approximated calculations of $\mathbf{d}_t^T \mathbf{s}_t$. The experiments indicated a minor speedup in favor of the exact calcuation. Equation (C.15) is in many cases a good approximation but can, however, be numerical unstable even when high precision arithmetic is used. If the relative error of $E'(\mathbf{w}_t)$ is ε then the relative error of equation (C.15) can be as high as $\frac{2\varepsilon}{\sigma_t}$ [Ralston et al. 78]. So the relative error gets higher when σ_t is lowered. We refer to [Møller 93a] for a detailed description of SCG. For a stochastic version of SCG especially designed for training on large, redundant training sets, see also [Møller 93b].

C.5.2 Eigenvalue estimation

A recent gradient descent learning algorithm proposed by Le Cun, Simard and Pearlmutter involves the estimation of the eigenvalues of the Hessian matrix. We will give a brief description of the ideas in this algorithm mainly in order to explain the use of the eigenvalues and the technique to estimate them. We refer to [Le Cun et al. 93] for a detailed description of this algorithm.

Assume that the Hessian $H(\mathbf{w}_t)$ is invertible. We then have by the spectral theorem from linear algebra that $H(\mathbf{w}_t)$ has N eigenvectors that forms an orthogonal basis in $\Re^N$ [Horn and Johnson 85]. This implies that the inverse of the Hessian matrix $H(\mathbf{w}_t)^{-1}$ can be written in the form

$$H(\mathbf{w}_t)^{-1} = \sum_{i=1}^{N} \frac{\mathbf{e}_i \mathbf{e}_i^T}{|\mathbf{e}_i|^2 \lambda_i} \ , \tag{C.16}$$

where λ_i is the i'th eigenvalue of $H(\mathbf{w}_t)$ and $\mathbf{e}_i$ is the corresponding eigenvector. Equation (C.16) implies that the search directions $\mathbf{d}_t$ of the Newton algorithm [Fletcher 75] can be written as

$$\mathbf{d}_t = \Leftrightarrow H(\mathbf{w}_t)^{-1}\mathbf{G}(\mathbf{w}_t) = \Leftrightarrow \sum_{i=1}^{N} \frac{\mathbf{e}_i \mathbf{e}_i^T}{|\mathbf{e}_i|^2 \lambda_i} \mathbf{G}(\mathbf{w}_t) = \Leftrightarrow \sum_{i=1}^{N} \frac{\mathbf{e}_i^T \mathbf{G}(\mathbf{w}_t)}{|\mathbf{e}_i|^2 \lambda_i} \mathbf{e}_i \ , \tag{C.17}$$

where $\mathbf{G}(\mathbf{w}_t)$ is the gradient vector. So the Newton search direction can be interpreted as a sum of projections of the gradient vector onto the eigenvectors weighted with the inverse of the eigenvalues. To calculate all eigenvalues and corresponding eigenvectors costs in $O(N^3)$ time which is infeasible for large N. Le Cun et al. argues that only a few of the largest eigenvalues and the corresponding eigenvectors is needed to achieve a considerable speed up in learning. The idea is to reduce the weight change in directions with large curvature, while keeping it large in all other directions. They choose the search direction to be

$$\mathbf{d}_t = \Leftrightarrow \left(\mathbf{G}(\mathbf{w}_t) \Leftrightarrow \frac{\lambda_{k+1}}{\lambda_1} \sum_{i=1}^{k} \frac{\mathbf{e}_i^T \mathbf{G}(\mathbf{w}_t)}{|\mathbf{e}_i|^2} \mathbf{e}_i \right) , \tag{C.18}$$

where i now runs from the largest eigenvalue λ_1 down to the k'th largest eigenvalue λ_k. The eigenvalues of the Hessian matrix are the curvatures in the direction of the corresponding eigenvectors. So Equation (C.18) reduces the component of the gradient along the directions with large curvature. See also [Le Cun et al. 91] for a discussion of this. The learning rate can now be increased with a factor of $\frac{\lambda_1}{\lambda_{k+1}}$, since the components in directions with large curvature has been reduced with the inverse of this factor.

The largest eigenvalue and the corresponding eigenvector can be estimated by an iterative process known as the *Power method* [Ralston et al. 78]. The Power method can be used successively to estimate the k largest eigenvalues if the components in the directions of already estimated eigenvectors are substracted in the process. Below we show an algorithm for estimation of the i'th eigenvalue and eigenvector. The Power method is here combined with the *Rayleigh quotient technique* [Ralston et al. 78]. This can accelerate the process considerably.

Choose an initial random vector $\mathbf{e}_i^0$. Repeat the following steps for $m = 1, \ldots, M$, where M is a small constant:

$$\mathbf{e}_i^m = H(\mathbf{w}_t)\mathbf{e}_i^{m-1} \ , \quad \mathbf{e}_i^m = \mathbf{e}_i^m \Leftrightarrow \textstyle\sum_{j=1}^{i-1} \frac{\mathbf{e}_j^T \mathbf{e}_i^m}{|\mathbf{e}_j|^2} \mathbf{e}_j$$

$$\lambda_i^m = \frac{(\mathbf{e}_i^{m-1})^T \mathbf{e}_i^m}{|\mathbf{e}_i^{m-1}|^2} \ , \quad \mathbf{e}_i^m = \frac{1}{\lambda_i^m} \mathbf{e}_i^m \ .$$

λ_i^M and $\mathbf{e}_i^M$ are respectively the estimated eigenvalue and eigenvector. Theoretically it would be enough to substract the component in the direction of already estimated eigenvectors once, but in practice roundoff errors will generally introduce these components again.

Le Cun et al. approximates the term $H(\mathbf{w}_t)\mathbf{e}_i^m$ with a one sided differencing as shown in equation (C.15). Now this term can be calculated exactly by use of the algorithm described in the last sections.

C.6 Conclusion

This paper has presented an algorithm for the exact calculation of the product of the Hessian matrix of error functions and a vector. The product is calculated without ever explicitly calculating the Hessian matrix itself. The algorithm has $O(PN)$ time and $O(N)$ memory requirements, where P is the number of patterns and N is the number of variables.

The relevance of this algorithm has been demonstrated by showing possible improvements in two different learning techniques, the scaled conjugate gradient learning algorithm and an algorithm recently proposed by Le Cun, Simard and Pearlmutter.

Acknowledgements

It has recently come to the authors knowledge that the same algorithm has been derived independently and at approximately the same time by Barak Pearlmutter, Department of Computer Science and Engineering Oregon Graduate Institute [Pearlmutter 93]. Thank you to Barak for his nice and immediate recognition of the independence of our work.

I would also like to thank Wray Buntine, Scott Fahlman, Brian Mayoh and Ole Østerby for helpful advice. This research was supported by a grant from the Royal Danish Research Council. Facilities for this research were provided by the National Science Foundation (U.S.) under grant IRI-9214873. All opinions, findings, conclusions and recommendations in this paper are those of the author and do not necessarily reflect the views of the Royal Danish Research Council or the National Science Foundation.

Appendix D

Adaptive Preconditioning of the Hessian Matrix

The paper [Møller 93d] was written in the spring of 1993 and has been submitted to Neural Computation and published as a technical report at DAIMI, Aarhus University. The following is a non modified version of this paper.

D.1 Abstract

The convergence rate of gradient learning methods depends on the condition number of the Hessian matrix. The smaller the condition number the faster convergence can be expected. A well-known technique to improve convergence in conjugate gradient algorithms is to precondition the Hessian before learning. This usually involves an incomplete LU factorization of the Hessian. This technique is very time consuming and can only be applied to positive definite Hessian matrices.

This paper propose an adaptive scheme to precondition the Hessian, which involves minimization of an additional error function during learning. The scheme preconditions the Hessian in the direction of low condition number without too much additional calculation work per iteration. The adaptive peconditioning is combined with gradient descent and the scaled conjugate gradient method. Experiments indicate a significant increase of convergence for gradient descent and a minor speedup for scaled conjugate gradient.

D.2 Introduction

The Hessian matrix of feed-forward neural network error functions plays an important role in learning since it contains all the second-order information about the error. Although exact formulae for the calculation of the Hessian exist [Buntine and Weigend 91a], [Bishop 92], they have not been widely used because of the large time and memory requirements involved in the calculation. Several researchers have now shown that Hessian information can be derived and used in a way that avoids the large computation and memory requirements. See for example [Le Cun et al. 93], [Møller 93c] and [Pearlmutter 93]. Møller and Pearlmutter have independently shown that the product of the Hessian and a vector can be calculated in the same order of time as a gradient calculation.

There is a strong correlation between the condition number κ of the Hessian matrix and the convergence rate of learning algorithms like gradient descent or conjugate gradient. κ is given by the ratio of the largest and smallest eigenvalue of the Hessian [Gill et al. 81]. If κ is high the Hessian is ill-conditioned and the convergence rate is expected to be slow. The conditioning of the Hessian of least mean square error functions when applied to linear feed-forward networks has been well studied in the literature [Widrow and Stearns 85], [Orfanidis 90], [Le Cun et al. 91]. Strong correlation among the components of the input vectors yields several large eigenvalues to the Hessian matrix which gives high κ and slow convergence. If the input vectors are decorrelated by a preprocessing scheme, then the eigenvalues are equalized which gives a faster convergence. Orfanidis generalizes this idea to multi-layer feed-forward networks by inserting preprocessors at each layer in the network.

Another approach is to reduce the effect of some of the ill-conditioned components in the Hessian matrix. Le Cun et al. describes an approach where components along eigenvectors with large corresponding eigenvalues are filtered out of the gradient update direction [Le Cun et al. 93]. This allows a larger learning rate to be applied, since directions with large curvature are reduced.

Both approaches are examples of a more general approach of preconditioning the Hessian. In conjugate gradient algorithms the ideas of preconditioning is well known [Fletcher 75], [Gill et al. 81]. The preconditioning scheme usually involves a transformation of the Newtonian linear system by means of a *preconditioning matrix*, so that the transformed Hessian is well-conditioned. The traditional calculation of the preconditioning matrix is, however, very time consuming for large scale problems and is for that reason not suitable for neural network problems. We describe another approach to estimate the preconditioning matrix, which involves successive adaptation of the matrix. The transformation considered in this paper is a simple scaling of variables, i.e., the preconditioning matrix is diagonal. Other transformations could easily be applied.

D.3 Notation

The networks we consider are multilayered feed-forward neural networks with arbitrary connectivity. The network $\aleph$ consist of nodes n^l_m arranged in layers $l = 0, \ldots, L$. The number of nodes in a layer l is denoted N_l. In order to be able to handle the arbitrary connectivity we define for each node n^l_m a set of *source nodes* and a set of *target nodes*.

$$\begin{aligned} S^l_m &= \left\{ n^r_s \in \aleph \,|\, n^r_s \text{ connects to } n^l_m, \;\; r < l, \;\; 1 \le s \le N_r \right\} \\ T^l_m &= \left\{ n^r_s \in \aleph \,|\, n^l_m \text{ connects to } n^r_s, \;\; r > l, \;\; 1 \le s \le N_r \right\} \end{aligned} \tag{D.1}$$

The training set accociated with network $\aleph$ is

$$\left\{ (u^0_{ps}, s = 1, \ldots, N_0, \;\; t_{pj}, j = 1, \ldots, N_L), p = 1, \ldots, P \right\} \tag{D.2}$$

The output from a node n^l_m when a pattern p is propagated through the network is

$$u^l_{pm} = f(v^l_{pm}) \text{ , where } v^l_{pm} = \sum_{n^r_s \in S^l_m} w^{lr}_{ms} u^r_{ps} + w^l_m , \tag{D.3}$$

and w^{lr}_{ms} is the weight from node n^r_s to node n^l_m. w^l_m is the usual *bias* of node n^l_m. $f(v^l_{pm})$ is an appropriate activation function, e.g., hyperbolic tangens. The net-input v^l_{pm} is chosen to be the usual weighted linear summation of inputs. The calculations to be made could, however, easily be extended to other definitions of v^l_{pm}. Let an error function $E(\mathbf{w})$ be

$$E(\mathbf{w}) = \sum_{p=1}^{P} E_p(u^L_{p1}, \ldots, u^L_{pN_L}, t_{p1}, \ldots, t_{pN_L}) \ , \tag{D.4}$$

where $\mathbf{w}$ is a vector containing all weights and biases in the network, and E_p is some appropriate error measure associated with pattern p from the training set. Coordinates of vectors and matrices will depending on the context also be referred to by the simpler notation $[\mathbf{w}]_i$ and $[A]_{ij}$.

D.4 Condition number and convergence rates

In this section we give a brief description of the correspondence between condition number and convergence rate for gradient descent and conjugate gradient algorithms. We assume that the error function $E(\mathbf{w})$ is locally quadratic so that

$$E(\mathbf{w} + \mathbf{h}) \approx E(\mathbf{w}) + \mathbf{h}^T E'(\mathbf{w}) + \frac{1}{2}\mathbf{h}^T E''(\mathbf{w})\mathbf{h} \ . \tag{D.5}$$

The eigenvalues of the Hessian matrix $E''(\mathbf{w})$ are the curvatures in the direction of the corresponding eigenvectors. The error changes most rapidly in the direction of the eigenvector corresponding to the largest eigenvalue λ_{max} and most slowly in the direction of the eigenvector corresponding to the smallest eigenvalue λ_{min}. The condition number of the Hessian matrix $E''(\mathbf{w})$ is defined as

$$\kappa = \left| \frac{\lambda_{max}}{\lambda_{min}} \right| \ . \tag{D.6}$$

We assume for the time being that the Hessian is positive definite so that all the eigenvalues are positive. The spread of the eigenvalues defines the shape of the contours of equal error. When κ equals one, the contours are circular and the gradient descent direction points directly to the minimum. When κ is greater than one, the contours are elliptical and the gradient descent direction does not necessarily point towards the minimum [Jacobs 88]. In general the higher the condition number is, the worse convergence can be expected of the gradient descent and the conjugate gradient algorithm. There is a direct link between the value of the condition number and the convergence rate of the algorithms. See [Concus et al. 76], [Axelsson 77] and [Aoki 71] for explanation of this. In the $k'th$ iteration of the gradient descent algorithm the relative error is bounded by

$$\frac{E(\mathbf{w}_k)}{E(\mathbf{w}_0)} \leq \left(\frac{\kappa \Leftrightarrow 1}{\kappa + 1} \right)^{2k} \ . \tag{D.7}$$

For the conjugate gradient algorithm we have

$$\frac{E(\mathbf{w}_k)}{E(\mathbf{w}_0)} \leq 4 \left(\frac{\sqrt{\kappa} \Leftrightarrow 1}{\sqrt{\kappa} + 1} \right)^{2k} \ . \tag{D.8}$$

The necessary number of iterations for gradient descent and conjugate gradient to reach a relative error of say ε is for large κ and small ε proportional to κ and $\sqrt{\kappa}$ respectively

[Axelsson 77]. For both algorithms is it clearly desirable to have the condition number as small as possible. For the conjugate gradient algorithm it is also true that the number of iterations needed to minimize (D.5) is proportional to the number of distinct eigenvalues of the Hessian [Gill et al. 81]. Seen from this perspective it is also desirable to have the condition number small.

Assume now, that the Hessian matrix is indefinite. The condition number defined in (D.6) does not give the same information about the conditioning of the Hessian, since there might be intermediate eigenvalues closer to zero. Instead of the smallest eigenvalue, we should now consider the eigenvalue closest to zero. If this eigenvalue is denoted λ_0, then a generalized definition of condition number could be

$$\kappa = \left| \frac{\lambda_{max}}{\lambda_0} \right| . \tag{D.9}$$

If the Hessian is invertible, then λ_0 is equal to the largest eigenvalue of the inverse. Notice, that this definition equals (D.6), when the Hessian is positive definite. The generalized condition number states something about the convergence to any stationary point. When the Hessian is indefinite, this will be saddle points.

D.5 Preconditioning

Minimization of (D.5) is equivalent to solving the linear system

$$E^{''}(\mathbf{w})\mathbf{h} = \Leftrightarrow E^{'}(\mathbf{w}) \tag{D.10}$$

The rate of convergence of gradient descent and conjugate gradient can significantly be improved if this system can be replaced by an equivalent system in which the Hessian has low condition number. The idea of preconditioning is to construct a transformation to have this effect on $E^{''}(\mathbf{w})$. The mostly used preconditioning scheme is *symmetric transformation*. The preconditioned system is then given by

$$A^T E^{''}(\mathbf{w}) A\mathbf{y} = \Leftrightarrow A^T E^{'}(\mathbf{w}) \ , \quad h = Ay \ , \tag{D.11}$$

where A is an $N \times N$ non-singular matrix. A should be chosen such that $A^T E^{''}(\mathbf{w})A$ has low condition number and is positive definite. Symmetric preconditioning corresponds to minimizing the error in the direction of A times the original search direction. One could use $A^T = L$, the Choleski factor of $E^{''}(\mathbf{w})$ [Fletcher 75]. This cost, however, $O(N^3)$ time to compute. A simpler choice is to choose A as a diagonal matrix, which is equivalent to a scaling of the variables. One particular choice could be to set the diagonal elements of A to the square root of the inverse of the diagonal elements in $E^{''}(\mathbf{w})$, then $A^T E^{''}(\mathbf{w})A$ would at least in some sense be close to unity since all diagonals would be equal to one.

A drawback with the symmetric transformation is that, if the Hessian is indefinite then so is the transformed matrix. This can be verified by

$$y^T A^T E^{''}(\mathbf{w}) A y = (Ay)^T E^{''}(\mathbf{w})(Ay) \ , \quad y \in \Re^N . \tag{D.12}$$

Since A is invertible the mapping given by (D.12) is equivalent with the mapping $x^T E^{''}(\mathbf{w})x$, $x \in \Re^N$. So if $E^{''}(\mathbf{w})$ is indefinite then $A^T E^{''}(\mathbf{w})A$ is also. Unfortunately, the Hessian

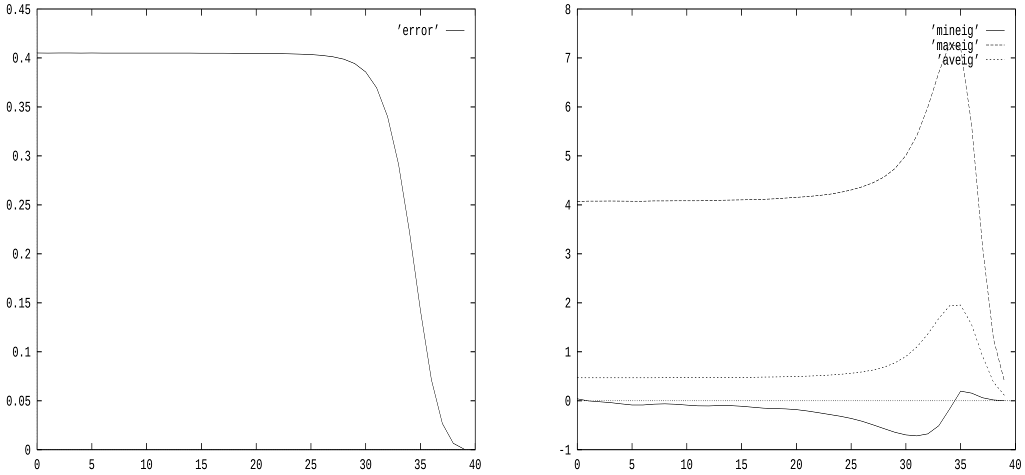

Figure D.1: A) The least mean square error curve on the XOR problem during learning with gradient descent. B) The largest, smallest and average eigenvalue of the Hessian during learning.

matrix of feed-forward network error functions is often indefinite. Figure D.1 shows an example. We observe that the Hessian is positive definite only very close to the minimum. Note also that the condition number is high during most of the minimization. This behaviour has also been observed by Kuhn and Watrous on various speech recognition problems [Kuhn and Watrous 93]. Even if the Hessian is indefinite, the symmetric transformation still makes sense, since the transformation can make it “less indefinite” by converting at least some of the negative eigenvalues to positive or by bringing the positive ones closer together.

The positive definiteness can be obtained by normalizing the linear system in (D.10) before the actual preconditioning. The normalization is

$$E''(\mathbf{w})^T E''(\mathbf{w})\mathbf{h} = \Leftrightarrow E''(\mathbf{w})^T E'(\mathbf{w}) \tag{D.13}$$

The preconditioned system is then given by

$$A^T E''(\mathbf{w})^T E''(\mathbf{w}) A\mathbf{y} = \Leftrightarrow A^T E''(\mathbf{w})^T E'(\mathbf{w})\ , \quad h = Ay\ . \tag{D.14}$$

The price for the positive definitenes is a significant increase of the condition number of the new matrix $E''(\mathbf{w})^T E''(\mathbf{w})$ compared to $E''(\mathbf{w})$. In fact it equals the square of the condition number of $E''(\mathbf{w})$ [Yang 92].

Another preconditioning scheme that also works on indefinite matrices is *nonsymmetric transformation* [Axelsson 80]. The linear system given by (D.10) is now transformed to

$$AE''(\mathbf{w})\mathbf{h} = \Leftrightarrow AE'(\mathbf{w})\ . \tag{D.15}$$

Again we may choose A in order to get a better conditioning of $AE''(\mathbf{w})$ than that of $E''(\mathbf{w})$. A should also be chosen such that $AE''(\mathbf{w})$ is positive definite. The price for the positive definitenes is in this scheme the loss of symmetry. The matrix $AE''(\mathbf{w})$ is not necessarily symmetric as was $E''(\mathbf{w})$. This is a problem for conjugate gradient like algorithms, because the symmetry implies that the residual vectors satisfy a three-term recursion, which is a powerful characteristic of conjugate gradient algorithms. Generalized conjugate gradient algorithms, which also work on unsymmetric systems have been proposed in the literature [Yang 92], [Axelsson 80]. The convergence properties of these algorithms have, however, not been fully determined.

In both the normalized symmetric preconditioning scheme and the nonsymmetric transformation, A is usually chosen as the inverse to some incomplete LU factorization of $E''(\mathbf{w})$ [Yang 92]. An incomplete LU factorization costs $O(N^3)$ operations. Since the Hessian matrix changes over time this factorization would have to be done several times during the minimization of the error. Considering the costs of this operation this is infeasible. As described in section 6, A can be estimated by an adaptive process during learning, which is much lower in cost.

The conclusion of this short survey of preconditioning schemes is, that no matter what scheme is chosen there is a price to be paid. For the symmetric transformation there was the lack of positive definitenes, for the normalized symmetric transformation a significant increase in initial condition number and for the nonsymmetric transformation the loss of symmetry. An additional disadvantage for the normalized symmetric scheme is also that the computational costs is higher than for the other two, since it involves double multiplication by the Hessian.

D.6 Gradient descent and conjugate gradient

This section describes how the preconditioning schemes are combined with gradient descent and conjugate gradient. The preconditioning scheme used in conjugate gradient is restricted to symmetric transformation. We define the following terms which vary in the different preconditioning schemes.

- Symmetric transformation: $\mathbf{r}_k = \Leftrightarrow A^T E'(\mathbf{w}_k)$, $G_k = A^T E''(\mathbf{w}_k) A$
- Normalized symmetric transformation:
$$\mathbf{r}_k = \Leftrightarrow A^T E''(\mathbf{w}_k)^T E'(\mathbf{w}_k) \ , \quad G_k = A^T E''(\mathbf{w}_k)^T E''(\mathbf{w}_k) A$$
- Nonsymmetric transformation: $\mathbf{r}_k = \Leftrightarrow E'(\mathbf{w}_k)$, $G_k = A^T E''(\mathbf{w}_k)$

Based on these definitions the gradient descent update with momentum is given by

$$\triangle \mathbf{w}_k = \eta A^T r_k + \alpha \triangle \mathbf{w}_{k-1} \ , \ \eta > 0 \ , \ 0 < \alpha < 1. \tag{D.16}$$

Once the preconditioning in gradient descent is effective the learning rate η can be increased, which speeds up the learning process. For that reason an adaptive learning rate scheme is needed to see the effect of the preconditioning. The learning rate in gradient descent is limited by the inverse of the largest eigenvalue λ_{max}. As described in the next section, λ_{max} is estimated in the preconditioning process, which makes it possible to use the learning rate $\eta = \frac{\eta_0}{\lambda_{max}}$, $\eta_0 > 0$. Whenever gradient descent is mentioned in the following it is combined with this adaptive learning rate scheme.[1]

[1] Some caution should be taken when applying this scheme. In some rear instances the estimate of λ_{max} may be negative which makes the learning rate negative. In this situation it is recommended to undo the step that resulted in a negative eigenvalue and update with a small positive learning rate.

The standard conjugate gradient algorithm combined with the symmetric preconditioning is as follows. The other two schemes are not the same but somewhat similar.

```
1. Select initial weight vector $\mathbf{w}_1$ ;
   $\mathbf{p}_1 = \mathbf{r}_1$ ;  $k = 1$ ;

2. $\alpha_k = \min_\alpha E(\mathbf{w}_k + \alpha A\mathbf{p}_k)$ ;

3. $\mathbf{w}_{k+1} = \mathbf{w}_k + \alpha_k A\mathbf{p}_k$ ;
   $\mathbf{r}_{k+1} = \Leftrightarrow A^T E'(\mathbf{w}_{k+1})$

4. if $(k \bmod N = 0)$ then
       $\mathbf{p}_{k+1} = \mathbf{r}_{k+1}$ ;
   else
       $\beta_k = \frac{|\mathbf{r}_{k+1}|^2 - \mathbf{r}_{k+1}^T \mathbf{r}_k}{|\mathbf{r}_k|^2}$ ;
       $\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$ ;

5. $k = k + 1$ ; terminate or go to 2.
```

The learning rate in step 2 is usually determined by a one dimensional line search, which can be very time consuming. In the scaled conjugate gradient algorithm (SCG) the learning rate is estimated by a scaling mechanism thus avoiding this line search. The learning rate is now given by

$$\alpha_k = \frac{\mathbf{p}_k^T \mathbf{r}_k}{\mathbf{p}_k^T G_k \mathbf{p}_k + \lambda_k \mathbf{p}_k^T A^T A \mathbf{p}_k} \; . \tag{D.17}$$

λ_k is a scaling parameter whose function is similar to the scaling parameter found in Levenberg-Marquardt methods [Fletcher 75]. λ_k is in each iteration raised or lowered according to how good the second order approximation is to the real error. The parameter Δ_k that measures the ratio between the real error change and the predicted quadratic error change is given by

$$\Delta_k = \frac{\left(\mathbf{p}_k^T G_k \mathbf{p}_k + \lambda_k \mathbf{p}_k^T A^T A \mathbf{p}_k\right)\left(E(\mathbf{w}_k) \Leftrightarrow E(\mathbf{w}_{k+1})\right)}{\left(\mathbf{p}_k^T \mathbf{r}_k\right)^2} \; . \tag{D.18}$$

An increase or decrease of the scaling parameter λ_k is controlled by the value of Δ_k. The preconditioning in SCG is done on the matrix $(E''(\mathbf{w}_k) + \lambda_k I)$, which is more often positive definite than the Hessian itself. The SCG algorithm combined with symmetric preconditioning has been used in the experiments described in section 7. Readers that are not familiar with the details of SCG are referred to [Møller 93a] and [Møller 93b].

D.7 Adaptive preconditioning

This section describes an approach to adapt the preconditioning matrix A during learning.

Let A be a diagonal matrix on the form $A = diag(\sigma(a_1), \sigma(a_2), \ldots, \sigma(a_N))$, where $\sigma(x)$ is a sigmoid function.[2] The preconditioning now corresponds to a scaling of variables. Throughout the paper A will be in diagonal form, though other forms could be considered.

[2] This particular form of the diagonals assure that A is invertible, positive definite and limited.

The adaptation of A should go in the direction of low condition number of G_k, as defined in equation (D.9). Considering that the Hessian is not necessarily positive definite, the best choice would be to adapt A to minimize the function

$$M(A) = \left| \frac{\lambda_{max}}{\lambda_0} \right| , \tag{D.19}$$

where λ_{max} is the largest eigenvalue of G_k and λ_0 is the eigenvalue closest to zero. λ_0 is, unfortunately, not easy to estimate in reasonable time, so another choice of $M(A)$ is required. One possibility is to ignore the indefinite Hessian matrices and choose $M(A)$ as the original definition of the condition number. Then $M(A)$ is

$$M(A) = \left| \frac{\lambda_{max}}{\lambda_{min}} \right| , \tag{D.20}$$

where λ_{max} and λ_{min} are the largest and smallest eigenvalue of G_k respectively. λ_{max} and λ_{min} can be estimated efficiently by the Power method as described below. The estimation of λ_{min} can, however, be a bit unstable. When the Hessian is indefinite, a minimization of (D.20) will force λ_{min} to be even more negative, while λ_{max} will be pulled towards zero. It is hard to predict the effect of such a preconditioning. Experiment indicate, that a better choice of $M(A)$ is

$$M(A) = \left| \frac{\lambda_{max}}{<\lambda>} \right| , \tag{D.21}$$

where $<\lambda>= \frac{1}{N} Tr(G_k)$ is the average eigenvalue and $Tr(G_k)$ is the trace of G_k. This function is also cheaper to compute since the eigenvalue estimation is more costly than the calculation of the trace.[3] It does, however, not solve the problem with the indefinite Hessian matrices. The gradient of $M(A)$ is

$$M'(A) = \; sign(\frac{\lambda_{max}}{<\lambda>}) \, \frac{1}{<\lambda>^2} \left(\frac{d\lambda_{max}}{dA} <\lambda> \Leftrightarrow \frac{d<\lambda>}{dA} \lambda_{max} \right) . \tag{D.22}$$

We will now show how to estimate (D.22) and use this estimation to perform gradient descent on A.

Depending on the preconditioning scheme used, the trace of G_k is

$$\begin{aligned}
Tr(G_k) &= \sum_{i=1}^{N} \sigma(a_i)^2 [E''(\mathbf{w}_k)]_{ii} \, , && \text{(symmetric)} && \text{(D.23)} \\
Tr(G_k) &= \sum_{i=1}^{N} \sigma(a_i)^2 \sum_{j=1}^{N} [E''(\mathbf{w}_k)]_{ij}^2 \, , && \text{(normalized symmetric)} && \text{(D.24)} \\
Tr(G_k) &= \sum_{i=1}^{N} \sigma(a_i) [E''(\mathbf{w}_k)]_{ii} \, , && \text{(nonsymmetric)} && \text{(D.25)}
\end{aligned}$$

The normalized symmetric preconditioning scheme involves off-diagonal terms of the Hessian which makes it costly to calculate. The diagonal elements of the Hessian $E''(\mathbf{w}_k)$, can be calculated by the following backward propagation formula [Le Cun 89].

[3] At least for the symmetric and the nonsymmetric transformation.

Corollary 3 *The diagonal elements of* $E^{''}(\mathbf{w}_k)$ *can be calculated by one backward propagation of the form:*

$$\frac{\partial^2 E}{(\partial w^{lh}_{mi})^2} = \frac{\partial^2 E}{(\partial v^l_{pm})^2}(u^h_{pi})^2 \ , \qquad \frac{\partial^2 E}{(\partial w^l_m)^2} = \frac{\partial^2 E}{(\partial v^l_{pm})^2} \ ,$$

where $\frac{\partial^2 E}{(\partial v^l_{pm})^2}$ *is*

- $\frac{\partial^2 E}{(\partial v^l_{pm})^2} = \sum_{n^r_s \in T^l_m} \left(f'(v^l_{pm})^2 (w^{rl}_{sm})^2 \frac{\partial^2 E}{(\partial v^r_{ps})^2} + f''(v^l_{pm}) w^{rl}_{sm} \frac{\partial E}{\partial v^r_{ps}} \right) \ , l < L \ ,$

  $\frac{\partial^2 E}{(\partial v^L_{pj})^2} = f'(v^L_{pj})^2 \frac{\partial^2 E_p}{(\partial u^L_{pj})^2} + f''(v^L_{pj}) \frac{\partial E_p}{\partial u^L_{pj}} \ , \quad 1 \leq j \leq N_L.$

The hard part of the adaptation is the estimation of the largest eigenvalue to G_k and its derivative. Let $\mathbf{e}^T_{max}$ be the eigenvector corresponding to λ_{max}. λ_{max} can then be written as

$$\lambda_{max} = \frac{\mathbf{e}^T_{max} G_k \mathbf{e}_{max}}{|\mathbf{e}_{max}|^2} \tag{D.26}$$

The derivative to equation (D.26) with respect to A turns out to be

$$\frac{d\lambda_{max}}{dA} = \frac{(\mathbf{e}_{max})^T \frac{dG_k}{dA} \mathbf{e}_{max}}{|\mathbf{e}_{max}|^2} \ , \tag{D.27}$$

So $\frac{d\lambda_{max}}{da_i}$ depends only on the change in G_k. For each preconditioning scheme the coordinates of (D.27) are

$$\frac{d\lambda_{max}}{da_i} = \frac{2\sigma'(a_i)[\mathbf{e}^T_{max}]_i [E^{''}(\mathbf{w}_k) A \mathbf{e}_{max}]_i}{|\mathbf{e}_{max}|^2} \qquad \text{(symmetric)} \tag{D.28}$$

$$\frac{d\lambda_{max}}{da_i} = \frac{2\sigma'(a_i)[\mathbf{e}^T_{max}]_i [E^{''}(\mathbf{w}_k)^T E^{''}(\mathbf{w}_k) A \mathbf{e}_{max}]_i}{|\mathbf{e}_{max}|^2} \qquad \text{(norm. symmetric)} \tag{D.29}$$

$$\frac{d\lambda_{max}}{da_i} = \frac{\sigma'(a_i)[\mathbf{e}^T_{max}]_i [E^{''}(\mathbf{w}_k) \mathbf{e}_{max}]_i}{|\mathbf{e}_{max}|^2} \qquad \text{(nonsymmetric).} \tag{D.30}$$

λ_{max} and $\mathbf{e}_{max}$ can be estimated by the Power method [Ralston et al. 78]. The method estimates the largest absolut eigenvalue and corresponding eigenvector. Unless the Hessian is extremely indefinite with a negative eigenvalue larger in absolute value than the largest positive one, this will be an estimate of λ_{max} and $\mathbf{e}_{max}$. The method is as follows.

- Choose an initial random vector $\mathbf{e}^0_{max}$.

- Repeat the following steps for $m = 1, \ldots, M, \ M > 0$:

  - $\mathbf{e}^m_{max} = G_k \mathbf{e}^{m-1}_{max}$,
  - $\lambda^m_{max} = \frac{(\mathbf{e}^{m-1}_{max})^T \mathbf{e}^m_{max}}{|\mathbf{e}^{m-1}_{max}|^2}$,
  - $\mathbf{e}^m_{max} = \frac{1}{\lambda^m_{max}} \mathbf{e}^m_{max}$.

λ^M_{max} and $\mathbf{e}^M_{max}$ are respectively the estimated eigenvalue and eigenvector. Similarly, the smallest eigenvalue and corresponding eigenvector can be estimated by applying the Power method on the matrix $(\lambda_{max} I \Leftrightarrow G_k)$. For the nonsymmetric preconditioning case, the eigenvalues might not be real. In this case we only operate on the real part of the eigenvalues. There has been some empirical evidence that positive real parts gives better convergence for conjugate gradient algorithms [Yang 92].

The term $G_k e^{m-1}_{max}$ in the Power method can be calculated as follows. As shown in [Møller 93c] and [Pearlmutter 93], the Hessian times a vector can be calculated exactly in $O(PN)$ time without ever having to calculate the Hessian matrix itself. We shortly summarize this result.

Corollary 4 *The product $E^{''}(\mathbf{w}_k)\mathbf{d}$, where $\mathbf{d}$ is a vector, can be calculated by one forward and one backward propagation. The forward propagation is:*

$$\varphi^l_{pm} = \sum_{n^r_s \in S^l_m} \left(d^{lr}_{ms} u^r_{ps} + w^{lr}_{ms} f'(v^r_{ps}) \varphi^r_{ps} \right) + d^l_m \quad ,l > 0 \;, \qquad \varphi^0_{pi} = 0 \;\; ,1 \le i \le N_0 .$$

The backward propagation is:

$$[E^{''}(\mathbf{w}_k)\mathbf{d}]^{lh}_{mi} = \delta^l_{pm} f'(v^h_{pi}) \varphi^h_{pi} + (\mu^l_{pm} + \beta^l_{pm}) u^h_{pi} \;, \qquad [E^{''}(\mathbf{w}_k)\mathbf{d}]^l_m = \mu^l_{pm} + \beta^l_{pm} \;,$$

where μ^l_{pm} δ^l_{pm} and β^l_{pm} are given by

- $\mu^l_{pm} = f'(v^l_{pm}) \sum_{n^r_s \in T^l_m} w^{rl}_{sm} \mu^r_{ps} \quad ,l < L \;,$

  $\mu^L_{pj} = \left(f'(v^L_{pj})^2 \frac{\partial^2 E_p}{(\partial u^L_{pj})^2} + f''(v^L_{pj}) \frac{\partial E_p}{\partial u^L_{pj}} \right) \varphi^L_{pj} \quad ,1 \le j \le N_L .$

- $\delta^l_{pm} = f'(v^l_{pm}) \sum_{n^r_s \in T^l_m} w^{rl}_{sm} \delta^r_{ps} \quad ,l < L \;,$

  $\delta^L_{pj} = \frac{\partial E_p}{\partial v^L_{pj}} \quad ,1 \le j \le N_L .$

- $\beta^l_{pm} = \sum_{n^r_s \in T^l_m} \left(f'(v^l_{pm}) w^{rl}_{sm} \beta^r_{ps} + \left(d^{rl}_{sm} f'(v^l_{pm}) + w^{rl}_{sm} f''(v^l_{pm}) \varphi^l_{pm} \right) \delta^r_{ps} \right) \quad ,l < L \;,$

  $\beta^L_{pj} = 0 \quad ,1 \le j \le N_L$

It is easy to see that the calculation of $G_k e^{m-1}_{max}$ can be based on the results from corollary 4. An example is for the symmetric transformation where $G_k = A^T E^{''}(\mathbf{w}_k) A$. Following the order of the parentheses, $G_k e^{m-1}_{max}$ is calculated as $G_k = A^T (E^{''}(\mathbf{w}_k)(A e^{m-1}_{max}))$.

One iteration of the Power method costs $O(PN)$ time, which is in the same order as the calculation of the gradient to the error. Even if M is small this is an substantial amount of work in order just to adapt A once based on equation (D.22). The following observations helps to reduce this calculation work. The Hessian matrix usual changes slowly over time near local minima so the minimization of $M(A)$ does not need to be done for every update of the weights. Furthermore, it is enough to have a rough estimate of the largest eigenvalue to do adaptation of A. We redefine the Power method to be

- Choose an initial random vector $\mathbf{e}^0_{max}$.
- Repeat the following steps for $t = 1, \ldots, T$, $T > 0$:
    - Repeat the following steps for $m = 1, \ldots, M$, $M > 0$:

- $\mathbf{e}^m_{max} = G_k \mathbf{e}^{m-1}_{max}$,
- $\lambda^m_{max} = \frac{(\mathbf{e}^{m-1}_{max})^T \mathbf{e}^m_{max}}{|\mathbf{e}^{m-1}_{max}|^2}$,
- $\mathbf{e}^m_{max} = \frac{1}{\lambda^m_{max}} \mathbf{e}^m_{max}$,
- $\frac{d\lambda^M_{max}}{dA} = \frac{(\mathbf{e}^M_{max})^T \frac{dG_k}{dA} \mathbf{e}^M_{max}}{|\mathbf{e}^M_{max}|^2}$,
- $a_i = a_i \Leftrightarrow \mu_{it}\ sign(\frac{\lambda^M_{max}}{<\lambda>})\ \frac{1}{<\lambda>^2} \left(\frac{d\lambda^M_{max}}{da_i} < \lambda > \Leftrightarrow \frac{d<\lambda>}{da_i} \lambda^M_{max} \right)$, $\quad 1 \leq a_i \leq N.$

The individual learning rate μ_{it} for each a_i is updated by

$$\mu_{it} = \begin{cases} 1.1\ \mu_{it-1} & \text{if } \triangle a_i(t) \triangle a_i(t \Leftrightarrow 1) > 0 \\ 0.1\ \mu_{it-1} & \text{otherwise} \end{cases} \tag{D.31}$$

As an additional constraint, the a_i's are limited to be in the range $[\Leftrightarrow 6, 6]$ in order to keep the derivatives $\sigma'(a_i)$ away from zero. In practice the process is run simultanously with the updates of weights, so that A is updated say for every K weight updates. The extra time and memory requirements added to the learning algorithm per weight update is then $\frac{MT}{K} O(PN)$, which is in the same order as performing $\frac{MT}{K}$ gradient calculations more. The parameters T and M can often be set to small values. A configuration, that yields $\frac{MT}{K} = 1$ is not unusual.

D.8 Experiments

In this section the adaptive preconditioning schemes are tested on gradient descent and scaled conjugate gradient. We have decided not to go further with the normalized symmetric preconditioning scheme because of the disadvantages of this scheme. Disadvantages that included larger computation time and higher initial condition number of the Hessian than that of the other two schemes. The test problems used are the XOR problem, the parity 5 problem [Rumelhart et al. 86] and the two spirals problem [Lang and Witbrock 89]. In the end of the section we outline ideas how to combine preconditioning with on-line learning techniques.

The following terms were the same for all experiments. A *failure* is defined to be a run that exceeds a certain predefined high number of iterations. A run is terminated and considered *successful* when all outputs are within a margin of 0.8 from the targets.

D.8.1 Gradient descent

The symmetric and nonsymmetric preconditioning schemes combined with gradient descent with adaptive learning rate were tested 30 times on the XOR problem. The standard gradient descent, gradient descent with adaptive learning rate and a new interesting method proposed by le Cun et al. were also tested. The method by le Cun et al. subtracts components along eigenvectors with large corresponding eigenvalues from the gradient direction [Le Cun et al. 91], [Le Cun et al. 93]. If only the component corresponding to the largest eigenvalue is substracted then the weight change is given by

$$\begin{aligned} \triangle \mathbf{w}_k \quad &= \quad \Leftrightarrow \eta \left(E'(\mathbf{w}_k) \Leftrightarrow (1 \Leftrightarrow \frac{<\lambda>}{\lambda_{max}}) \frac{\mathbf{e}^T_{max} E'(\mathbf{w}_k)}{|\mathbf{e}_{max}|^2} \mathbf{e}_{max} \right) + \ \alpha \triangle \mathbf{w}_{k-1} \ , \\ &\qquad \eta > 0 \ , \quad 0 < \alpha < 1 \ . \end{aligned} \tag{D.32}$$

	GD	AGD	SAGD	SAGD	NAGD	NAGD	RGD
param.	$\eta = 0.25$	$\eta_0 = 1$	$K = 1$ $T = 1$ $M = 1$ $\eta_0 = 1$ $\mu_0 = 1$	$K = 7$ $T = 14$ $M = 1$ $\eta_0 = 1$ $\mu_0 = 1$	$K = 1$ $T = 1$ $M = 1$ $\eta_0 = 1$ $\mu_0 = 1$	$K = 7$ $T = 14$ $M = 1$ $\eta_0 = 1$ $\mu_0 = 1$	$K = 1$ $T = 1$ $M = 1$ $\eta = 1$
mean	71.67	73.22	27.34	22.90	37.13	33.97	29.04
std.dev.	28.38	23.03	9.36	7.05	14.06	13.53	10.21
failures	3	3	4	2	1	2	2

Table D.1: Average results on the XOR problem. GD = standard gradient descent, AGD = adaptive gradient descent, SAGD= symmetric preconditioning + adaptive gradient descent, NAGD = nonsymmetric preconditioning + adaptive gradient descent, RGD = reduced gradient descent (Le Cun et al.'s method).

The average results are illustrated in table D.1. We observe that the adaptive learning rate scheme does not improve the convergence compared to a carefully tuned learning rate. Combined with the preconditioning schemes, however, there is a significant increase in convergence. The symmetric preconditioning scheme converge faster than the non-symmetric. We also observe that the better the meta-error function $M(A)$ is minimized the faster the convergence. However, the minimization of $M(A)$ takes time so there is a trade-off between the gain of convergence and the increase in computation time per weight update. The result of the method by le Cun et al. is comparable with the symmetric preconditioning.

Figure D.2 shows an example of a run with SAGD. This is the same run as the one illustrated in figure D.1 with GD. The largest eigenvalue in figure D.2 drops rapidly in the first few iterations, while the largest eigenvalue in figure D.1 is constant until just before termination. SAGD gets faster through the flat plateau in the beginning of the minimization because of this rapid decrease of the largest eigenvalue.

Since there seems to be no advantage in using the normalized preconditioning scheme compared to the symmetric scheme, we now concentrate on the symmetric preconditioning scheme only. GD and SAGD were tested 20 times on the parity 5 problem with a 3-layer network with 5 hidden units. The average results are illustrated in table D.2. SAGD converges faster than AGD, but the number of failures seems to be very sensitive to the initial configuration of K, T and M. The first configuration with frequent and relatively many updates of A yields results, where SAGD fails in 8 out of 20 runs, while another configuration with not as frequent updates yields results with 6 failures. A characteristic error curve for a failure run of SAGD is illustrated in figure D.3. We observe, that the error drops more rapidly for SAGD than for AGD, i.e., the weights gets faster through the very flat region in the beginning, but then the weights converges to what turns out to be a saddle point. So the preconditioning works well in the beginning, but because the Hessian is indefinite, the preconditioning can force the algorithms to converge to a stationary point, which is not a minimum.[4] This is of course unfortunate. One way to avoid this behavior is to tune the initial parameters K, T and M or to turn the

[4] This can also happen when no preconditioning is applied, but the preconditioning seems to make this more likely to happen.

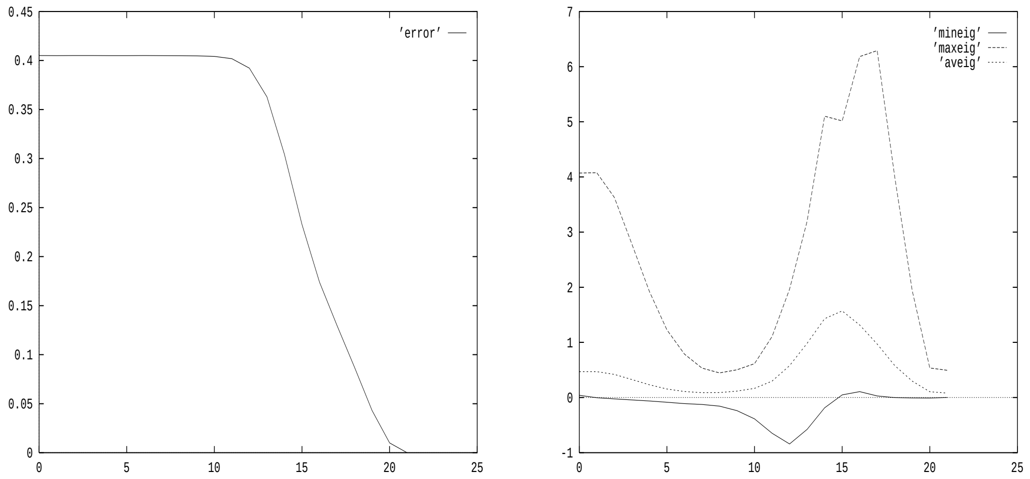

Figure D.2: A) The least mean square error curve on the XOR problem during learning with SAGD. B) The largest, smallest and average eigenvalue of the Hessian during learning.

	AGD	SAGD	SAGD	SAGD2
param.	$\eta_0 = 1$	$K = 20$ $T = 20$ $M = 2$ $\eta_0 = 1$ $\mu_0 = 10$	$K = 40$ $T = 10$ $M = 4$ $\eta_0 = 1$ $\mu_0 = 10$	$K = 20$ $T = 20$ $M = 2$ $\eta_0 = 1$ $\mu_0 = 10$ $\gamma = 1e^{-4}$
mean	3121	212	226	214
std.dev.	2640	123	133	89
failures	5	8	6	4

Table D.2: Average results on the parity 5 problem. GD = standard gradient descent, SAGD= symmetric preconditioning + adaptive gradient descent, SAGD2 = symmetric preconditioning in flat regions + adaptive gradient descent.

preconditioning off as soon as the weights have left the flat regions and then initialize the preconditioning matrix to the identity matrix. If we define the parameter Δ_k as

$$\Delta_k = \left| \frac{E(\mathbf{w}_k) \Leftrightarrow E(\mathbf{w}_{k-1})}{E(\mathbf{w}_{k-1})} \right| , \tag{D.33}$$

then the preconditioning is only applied when $\Delta_k < \gamma$, where γ is a small constant. The result of this strategy is also illustrated in table D.2 in the last coloum and in figure D.3. Again we see a very fast and reliable convergence towards a minimum.

D.8.2 Scaled conjugate gradient

The symmetric preconditioning scheme combined with SCG was tested on the XOR problem. Table D.3 shows the average results. Again we observe an increase in convergence

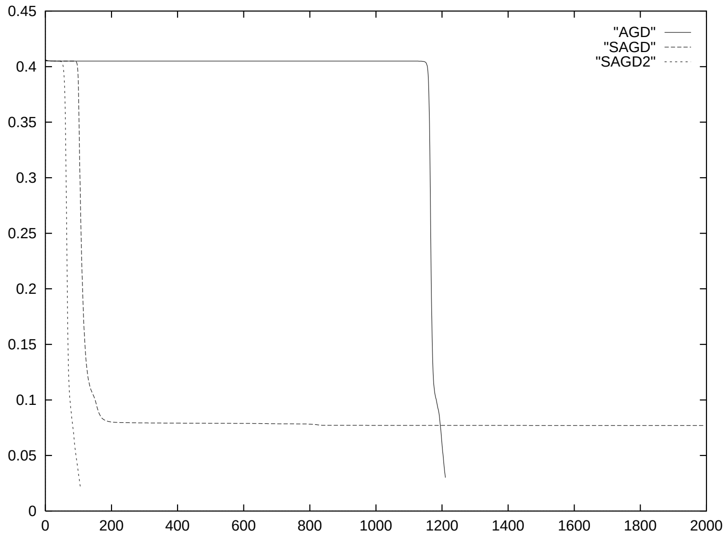

Figure D.3: Error curves for AGD, SAGD and SAGD2 on the parity 5 problem. The run of SAGD fails because of convergence to a saddle point.

	SCG	PSCG	PSCG	PSCG2

param.		K=10
		T=10
		M=1
		$\mu_0 = 10$
			K=1
			T=1
			M=1
			$\mu_0 = 10$
				K=40
				T=20
				M=3
				$\mu_0 = 10$
				$\gamma = 5e^{-4}$

mean	16.54	15.03	14.56	13.77
std.dev.	3.80	4.33	3.61	2.18
failures	4	4	2	3

Table D.3: Average results on the XOR problem. SCG = scaled conjugate gradient, PSCG = symmetric preconditioning + scaled conjugate gradient, PSCG2 = symmetric preconditioning in flat regions + scaled conjugate gradient.

when the Hessian is preconditioned.

SCG and PSCG was tested 20 times on the parity 5 problem using the same initial weights and architecture as in the last section. The average results are illustrated in table D.4. As observed in the last section, preconditioning during the whole minimization yields more failures. Stopping the preconditioning after the weights has left the flat region helps, also as noticed above. We do, however, not observe the same significant increase of convergence as with SAGD2.

SCG, PSCG and PSCG2 were run 10 times on the two-spirals problem, the task of distinguishing between two interwined spirals [Lang and Witbrock 89]. The problem is known to be hard for neural networks to solve and is a useful benchmark test. The network architecture used in this experiment was a 3 hidden layer network with 5 hidden units in each layer and shortcut connections from the input layer and upwards. The average results are illustrated in table D.5. PSCG2 converges faster than the other two, but only barely enough to justify the extra computational effort per iteration.

It is not clear why the same dramatic increase in convergence is not obtained, when

	SCG	PSCG	PSCG2

param.		K=40
		T=10
		M=4
		$\mu_0 = 10$
			K=20
			T=20
			M=2
			$\mu_0 = 10$
			$\gamma = 5e^{-4}$

mean	125	80	97
std.dev.	68	42	58
failures	4	6	4

Table D.4: Average results on the parity 5 problem. SCG = scaled conjugate gradient, PSCG = symmetric preconditioning + scaled conjugate gradient, PSCG2 = symmetric preconditioning in flat regions + scaled conjugate gradient.

	SCG	PSCG	PSCG2

param.		K=40
		T=10
		M=4
		$\mu_0 = 10$
			K=20
			T=10
			M=2
			$\mu_0 = 10$
			$\gamma = 5e^{-4}$

mean	1053	1197	855
std.dev.	546	586	257
failures	2	2	1

Table D.5: Average results on the two-spirals problem. SCG = scaled conjugate gradient, PSCG = symmetric preconditioning + scaled conjugate gradient, PSCG2 = symmetric preconditioning in flat regions + scaled conjugate gradient.

the preconditioning is combined with conjugate gradient, as when combined with gradient descent. It might be, that the change of A interferes too much with the conjugate directions.

D.8.3 On-line preconditioning

Recently there has been some discussion about the efficiency of off-line and on-line learning techniques. Whenever learning problems are characterized by large redundant training sets the on-line technique seems to be the most efficient. See for example [Battiti 92], [Le Cun et al. 93] or [Møller 93b]. In [Møller 93b] a technique to combine conjugate gradient learning methods with on-line learning is described. In this section we briefly describe how to combine the preconditioning approach with on-line learning. The main idea is to “relax” the preconditioning by replacing terms of the form $E''(\mathbf{w})\mathbf{d}$ with a running average like

$$E''(\mathbf{w})\mathbf{d} = \gamma E_p''(\mathbf{w})\mathbf{d} + (1 \Leftrightarrow \gamma)E''(\mathbf{w})\mathbf{d}\ ,\ \ 0 < \gamma < 1, \qquad \text{(D.34)}$$

where $E_p''(\mathbf{w})$ is the Hessian corresponding to pattern number p. Le Cun et al. have tried such a relaxation combined with the above RGD method with success on some handwritten digit recognition problems [Le Cun et al. 93].

D.9 Conclusion

This paper has proposed a new method to improve convergence of supervised learning algorithms. The method is based on an adaptive scheme to precondition the Hessian. While there is only a minor speed up for the scaled conjugate gradient algorithm, the method speeds up learning considerably for gradient descent and can improve the convergence through flat regions of weight space by several orders of magnitude. Adaptive preconditioning can be used in off-line mode as well as on-line mode and costs in the order $O(PN)$ per epoch to employ. If the preconditioning is only applied in flat regions of weights space then this computational overhead is even less.

The method should in principal work for any data-fit problem, which means that it is relevant in a broader context than the neural network field.

Acknowledgements

I would like to thank Ole Østerby, Ole Caprani and Brian Mayoh for helpful advice. This research was supported by a grant from the Royal Danish Research Council. All opinions, findings, conclusions and recommendations in this paper are those of the author and do not necessarily reflect the views of the Royal Danish Research Council.

Appendix E

Improving Network Solutions

The paper [Møller and Fahlman 93] is a collaboration work with Scott Fahlman, CMU, done in the period September 1992 to March 1993, while the author visited School of Computer Science, Carnegie Mellon University. The following is a modified version of this paper.

E.1 Abstract

Through a comparison study of two learning algorithms we propose ways to improve network solutions with respect to convergence and generalization. The Quickprop algorithm and the Scaled Conjugate Gradient algorithm are compared empirically. SCG is significantly faster than QP and contains no problem-dependent parameters. However, SCG ends more often in suboptimal solutions, with not as many correct classifications as QP. This is due to characteristics of the least square error function which is ill-suited for many neural network training problems. We impose appropriate constraints on network solutions by modification of error function. The new error functions yields network solutions that can improve convergence and generalization.

E.2 Introduction

When evaluating feed-forward neural network solutions it is necessary to consider the convergence rate to- and the generalization ability of the network solutions found. It is often argued that a very fast convergence yields bad generalization. This is not necessarily true. If proper constraints are imposed on the network solutions then fast convergence and good generalization can be obtained. However, the addition of too hard constraints can have a negative influence on the convergence rate since this severely limits the acceptable paths down to the solutions. Arguments for constraining network solutions can be found in Regularization theory [Hastie and Tibshirani 90], Bayesian inference [Buntine and Weigend 91b] and Minimum Description Length methods [Rissanen 84].

In this paper we impose constraints by modification of error function. Both convergence rate and generalization depends heavily on the error function used in the learning process. Error functions like least square and cross-entropy, are known to be Bayes optimal in the sense that minimization with these functions produce solutions that approach the greatest lower bound on generalization error as the training set approaches infinity. But when the training set is small this approximation can be poor [Buntine and Weigend 91b], [Wan 90], [Gish 90], [Schaffer 92]. So especially when the amount of data available is sparse it is necessary to impose constraints on the network solutions. This is in a Bayesian perspective the same as choosing appropriate priors which is strongly related to penalty terms or regularizers in statistical literature.

There are several ways to constrain network solutions by modification of error function, one way is to demand that all patterns are classified correctly. Another would be to demand a smooth distribution of errors, i.e., a low variance of absolute errors. Through a comparison study of the Scaled Conjugate Gradient algorithm (SCG) [Møller 93a] and the Quickprop algorithm (QP) [Fahlman 89] we propose two new error functions that empirically can be shown to improve generalization. The comparison is interesting in it self since both algorithms in earlier studies have shown to be efficient.

E.3 Comparison of two efficient learning algorithms

It is widely recognized that the class of conjugate gradient algorithms are well suited for learning algorithms because of their ability to gain second order information without too much calculation work [Watrous 87], [Parker 85], [Battiti 92]. One, the Scaled Conjugate Gradient algorithm, has especially low calculation costs, and has for that reason shown to be efficient for supervised learning. Another efficient algorithm widely used is the Quickprop algorithm, which also gains second order information at low costs. This section presents a comparison study of these two algorithms.

E.3.1 The Quickprop algorithm

The Quickprop algorithm developed by Fahlman is based on the Newton algorithm [Fletcher 75]. In order to avoid the time and space consuming calculations involved with the Newton algorithm two approximations are made. The Hessian matrix is approximated by ignoring all non-diagonal terms making the assumption that all the weights are independent. Each term in the diagonal is approximated by a one sided difference formula given by

$$\frac{d^2E(w)}{dw^2} \approx \frac{E'(w_t) \Leftrightarrow E'(w_{t-1})}{w_t \Leftrightarrow w_{t-1}} \tag{E.1}$$

where E(w) is the error and w_t is a given weight at time step t. $\frac{d^2E(w)}{dw^2}$ can actually be calculated precisely with a little more calculation work [Le Cun 89]. The weight update for the Quickprop algorithm is given by

$$\triangle w_t = \Leftrightarrow (\eta_t E'(w_t) + \alpha_t) \ , \tag{E.2}$$

where η_t is

$$\eta_t = \begin{cases} \eta_0 & E'(w_t)E'(w_{t-1}) > 0 \\ 0 & \text{otherwise} \end{cases}$$

and α_t is

$$\alpha_t = \begin{cases} \frac{w_t - w_{t-1}}{E'(w_t) - E'(w_{t-1})} E'(w_t) & \left| \frac{E'(w_t) - E'(w_{t-1})}{E'(w_{t-1})} \right| < \frac{\mu}{1+\mu} \\ \mu \triangle w_{t-1} & \text{otherwise} \end{cases}$$

The constant η_0 is similar to the learning rate in gradient descent. If $E'(w_t)E'(w_{t-1}) > 0$, i.e., the minimum of the quadratic has not been passed, a linear term is added to the quadratic weight change. On the other hand, if $E'(w_t)E'(w_{t-1}) \leq 0$, i.e., the minimum of the quadratic has been passed, only the quadratic weight change is used to go straight down to the minimum. μ is usually set equal to 2, which seems to work well in most applications.

The algorithm is usually used combined with a *primeoffset* term added to the first derivative of the sigmoid activation function. As noted later the use of primeoffset can influence the quality of the solutions found.

Despite the two very crude approximations the Quickprop algorithm has shown very good performance in practice. One drawback with the algorithm is, however, that the ϵ parameter is very problem dependent. We refer to [Fahlman 89] for a detailed description of the algorithm.

E.3.2 The Scaled Conjugate Gradient Algorithm

The Scaled Conjugate Gradient algorithm is a variation of a standard conjugate gradient algorithm. The major idea of conjugate gradient algorithms is that they up to second order produce non-interfering directions of search. This means that minimization in one direction $\mathbf{d}_t$ followed by minimization in another direction $\mathbf{d}_{t+1}$ imply that the error has been minimized over the whole subspace spanned by $\mathbf{d}_t$ and $\mathbf{d}_{t+1}$. The search directions are given by

$$\mathbf{d}_{t+1} = \Leftrightarrow E'(\mathbf{w}_{t+1}) + \beta_t \mathbf{d}_t \tag{E.3}$$

where $\mathbf{w}_t$ is a vector containing all weight values at time step t and β_t is

$$\beta_t = \frac{|E'(\mathbf{w}_{t+1})|^2 \Leftrightarrow E'(\mathbf{w}_{t+1})^T E'(\mathbf{w}_t)}{|E'(\mathbf{w}_t)|^2} \tag{E.4}$$

In the standard conjugate gradient algorithms the step size ϵ_t is found by a line search which can be very time consuming because this involves several calculations of the error and or the first derivative. In the Scaled Conjugate Gradient algorithm the step size is estimated by a scaling mechanism thus avoiding the time consuming line search. The step size is given by

$$\epsilon_t = \frac{\Leftrightarrow \mathbf{d}_t^T E'(\mathbf{w}_t)}{\mathbf{d}_t^T \mathbf{s}_t + \lambda_t |\mathbf{d}_t|^2} \tag{E.5}$$

where $\mathbf{s}_t$ is

$$\mathbf{s}_t = \frac{E'(\mathbf{w}_t + \sigma_t \mathbf{d}_t) \Leftrightarrow E'(\mathbf{w}_t)}{\sigma_t} \quad , 0 < \sigma_t \ll 1$$

ϵ_t is the step size that minimizes the second order approximation to the error function. $\mathbf{s}_t$ is a one sided difference approximation of $E''(\mathbf{w}_t)\mathbf{d}_t$. λ_t is a scaling parameter whose function is similar to the scaling parameter found in Levenberg-Marquardt methods [Fletcher 75]. λ_t is in each iteration raised or lowered according to how good the second order approximation is to the real error. The weight update formula is now given by

Seed	SCG Epoch	QP Epoch
1	372	8659
2	599	5570
3	2000*	10105
4	722	15182
5	1036	17615
6	2000*	12405
7	1217	15385
8	853	4528
mean	1099	11181
std.dev.	1121	4470

Table E.1: Results on the two spirals problem.

$$\triangle \mathbf{w}_t = \epsilon_t \mathbf{d}_t \tag{E.6}$$

The calculation work per iteration for SCG can be shown to be in the order of two times the calculation work for QP. SCG contains no problem dependent parameters. We refer to [Møller 93a] for a detailed description of SCG. For a stochastic version of SCG especially designed for training on large, redundant training sets, see also [Møller 93b]

E.3.3 Comparison

In this section SCG and QP are tested on the two spirals problem [Lang and Witbrock 89] and an artificial classification problem generated only for this purpose. The hyperbolic tangent was used as activation function through out all experiments. When comparing convergence rates the calculation work per iteration for the two algorithms is always taken into account.

Two spirals problem

A useful benchmark task for neural networks is the two spirals problem: the task of distinguishing between two interwined spirals. Lang and Withbrock tested QP against Back-Propagation [Rumelhart et al. 86] on this problem using a network of 3 hidden layers with 5 units per layer and short cut connections from the input layer and upwards.

Using the same architecture and training set as Lang and Witbrock 8 different tests with SCG and QP were run. The parameters for QP were: $\epsilon = .002$, $\mu = 1.75$, primeoffset $= 0$. Using an primeoffset term on this problem did not increase the efficiency of QP. The algorithms were terminated when all patterns were classified correctly using an error margin of 0.8.

The results are shown in table E.1. SCG yields an average speedup of 5.1 against QP. The mean number of epochs used by QP is of the same order as the results for QP reported by Lang and Witbrock. Notice that SCG in two of the cases fails to find a solution with a 100% classification. In these two cases the least square error is very small but 1-2 patterns are classified completely wrong.

Dim	SCG				QP			
	Error		Correct		Error		Correct	
	mean	std.dev.	mean	std.dev.	mean	std.dev.	mean	std.dev.
8	.028764	.008062	.984524	.004796	.010957	.007550	.997619	.003000
10	.029224	.0124810	.983750	.008718	.025807	.001000	.991875	.004243
12	.023387	.004624	.988303	.000007	.019548	.008756	.994643	.004140
14	.014461	.003822	.993208	.002057	.005670	.001886	.999292	.000867
16	.011717	.001818	.994690	.001180	.005214	.001210	.999410	.000723
18	.008912	.004110	.996120	.004823	.003772	.007745	1.	0.

Table E.2: Average results on artificial data.

Artificial data

To be able to see how SCG and QP compare on problems with varying input dimensions some artificial data was generated. For dimension N a set of 4N *centerpoints*, each a N-bit string, was randomly chosen. Around each centerpoint a set of 9 *distortions* was generated using a Gaussian distribution to determine whether to flip a bit or not. This gives a total of 40N patterns. Each centerpoint and its distortions were then randomly assigned to one out of two possible classes. SCG and QP were tested on dimension 8,10,12,14,16 and 18 running 5 different runs on each dimension using a 3 layer network with N hidden units. The algorithms were run for a constant number of epochs, QP twice as many as SCG in order to take the calculation work per epoch into account. The parameters for QP were: $\epsilon = 1/\text{noofpatterns}$, $\mu = 1.75$, primeoffset = 0.1.

The average performance is illustrated in table E.2. On average QP reaches a better solution than SCG both when regarding error and correct classification. Figure E.1 shows the average error curve and average classification curve during training for SCG and QP on dimension 12. The curves are basicly the same for all other dimensions. We conclude that SCG converge faster to a solution both with respect to error and correct classification but does not in average find as good a solution as QP. This is in fact because of QP's ability to use the primeoffset term. When QP is run without this term the average solutions found resembles these of SCG.

Adding a primeoffset to the derivative to the activation function means that the error derivative does not correspond to the real error. Since SCG needs the exact error derivative to estimate the error in a neighboring point it is not in any obvious way possible to use primeoffset with SCG.

E.4 Imposing constraints on network solutions

Clearly the bad solutions found by SCG are due to characteristics of the least square error function. The least square error function has many suboptimal solutions which are represented as very flat regions in weight space. Minimization of the least square error function does not imply minimization of misclassifications [Brady and Raghavan 88], [Makram-Ebeid et al. 89], [Yu and Simmons 90], [Hampshire 92]. Thus minimizing the least square error function often converge to suboptimal solutions with respect to the number of correct classifications.

One way to try to avoid these suboptimal solutions is like in the Quickprop algorithm

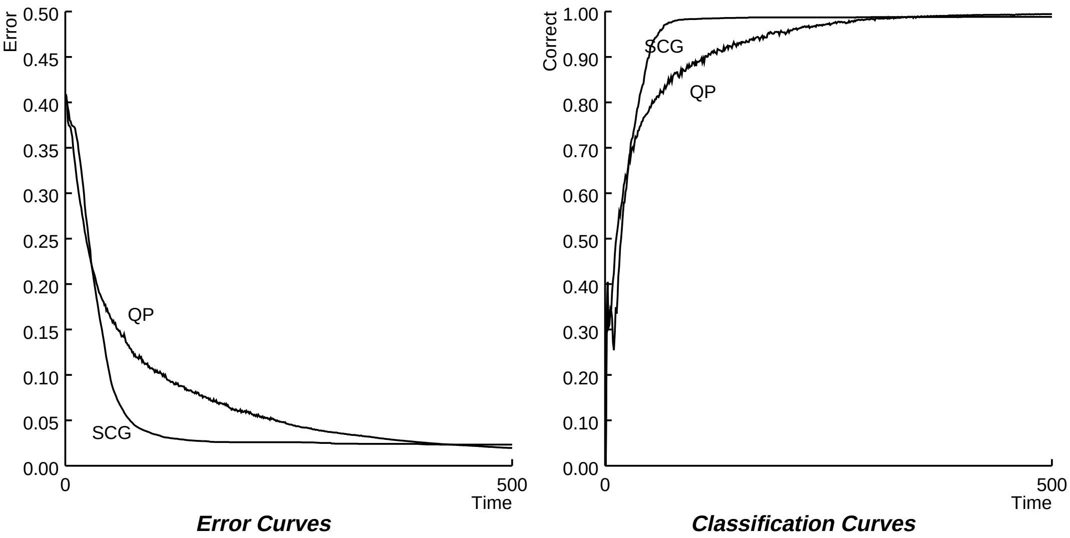

Figure E.1: Average error and classifications curves for SCG and QP on dimension 12.

to twist the first derivative to the sigmoid activation function by adding a primeoffset term. Another way is to strictly minimize the number of misclassifications. Hampshire defines such an approach that works for binary classification problems [Hampshire 92]. We present a more general approach that involves a soft minimization of misclassifications.

Since good solutions are characterized not only by low average error but also by having as many patterns with low error as possible, a good idea would be to include both terms in the error function. Several researchers have tried that. Mackram-Ebeid, Surat and Viola defines the following error function

$$E(\mathbf{w}) = \frac{1}{2}\sum_{p,j}\left\{\begin{array}{ll} \gamma(t_{pj} \Leftrightarrow o_{pj})^2 & \text{if } t_{pj}o_{pj} > 0. \\ (t_{pj} \Leftrightarrow o_{pj})^2 & \text{otherwise} \end{array}\right. \tag{E.7}$$

where t_{pj} and o_{pj} are respectively the desired target and the observed output at unit j when pattern p is presented. γ is gradually increased from 0 to 1 so that the function initially just focus on getting the sign of the outputs right and then later pays attention to the magnitude of the error [Makram-Ebeid et al. 89]. This approach only works for binary classification problems. Yu and Simmons defines another but similar error function

$$E(\mathbf{w}) = \frac{1}{2}\sum_{p,j}\left\{\begin{array}{ll} (t_{pj} \Leftrightarrow o_{pj})^2 & \text{if } |t_{pj} \Leftrightarrow o_{pj}| > \varepsilon \\ 0 & \text{otherwise} \end{array}\right. \tag{E.8}$$

where ε is a positive parameter which is decreased when the absolute value of *all* the partial errors are less than ε.

Common for these two approaches is that the error functions defined are not easy to use with more sophisticated algorithms like SCG because they are not differentiable. However, a way to fix this in Yu and Simmons error function would be to substract ε^2 in the first line.

Another approach is to define an error function that penalizes errors of large magnitude.

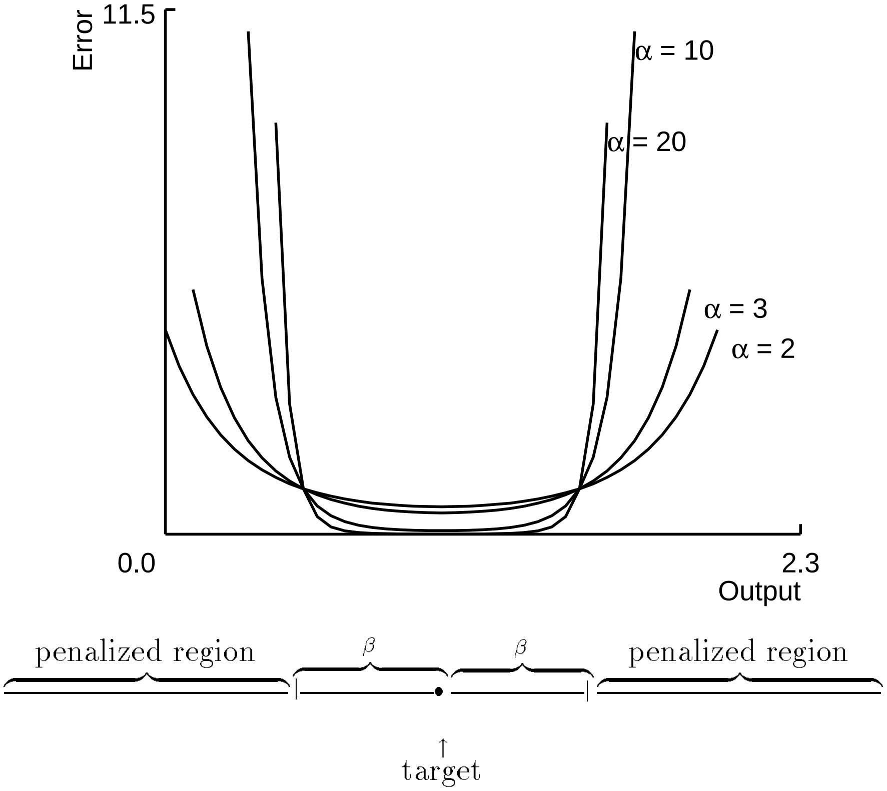

Figure E.2: The function of the α and β parameter.

$$E(\mathbf{w}) = \frac{1}{2}\sum_{p,j} e^{-\alpha(o_{pj}-t_{pj}+\beta)(t_{pj}+\beta-o_{pj})} \tag{E.9}$$

where α and β are positive parameters. The derivative to (E.9) with respect to a given o_{pj} is

$$\frac{dE(\mathbf{w})}{do_{pj}} = \Leftrightarrow\alpha(t_{pj} \Leftrightarrow o_{pj})e^{-\alpha(o_{pj}-t_{pj}+\beta)(t_{pj}+\beta-o_{pj})} \tag{E.10}$$

It is easy to see that the global minimum for (E.9) is when $t_{pj} = o_{pj}, \forall p, j$. The function of α and β is illustrated through figure E.2. β defines the width of the acceptable error around the desired target and α controls the steepness of the exponentially growing error in the penalized regions outside the interval. If α is small equation (E.10) resembles the derivative of the least square function. But the higher α gets the more active is the constraint imposed on the penalized regions. When no errors are in the penalized regions β is decreased, so that the outputs are pulled towards the targets. Note that the *exponential* error function indirectly balances the errors especially when α is large. A high α value gives large partial error derivatives inside the penalized regions and small partial error derivatives when outside the regions. So the higher the α value the more the errors will tend to arrange themselves around the boundary of the penalized regions. This gives a balanced distribution of the errors. Yu and Simmons shows that balancing the errors on the training set can improve the generalization ability of a network solution [Yu and Simmons 90].

Dim	SCG				QP				Speedup
	Epoch		Correct		Epoch		Correct		
	mean	std.dev.	mean	std.dev.	mean	std.dev.	mean	std.dev.	
8	76	3	1	0	249	28	1	0	1.6
10	112	19	1	0	758	111	1	0	3.4
12	109	10	1	0	763	180	1	0	3.5
14	77	12	1	0	499	66	1	0	3.2
16	75	1	1	0	551	80	1	0	3.7
18	78	7	1	0	423	43	1	0	2.7

Table E.3: Average results on artificial data using the exponential error function.

A more direct way of balancing errors is to minimize the variance of the magnitude of the errors. This can be done by adding the variance as a penalty term to an existing error function like least square.

$$E(\mathbf{w}) = \frac{1}{NP}\sum_{j}^{N}\sum_{p}^{P}(t_{pj}\Leftrightarrow o_{pj})^2 + \eta\frac{1}{NP}\sum_{j}^{N}\sum_{p}^{P}\Big((t_{pj}\Leftrightarrow o_{pj})^2 \Leftrightarrow \tfrac{1}{PN}\textstyle\sum_i^N\sum_q^P(t_{qi}\Leftrightarrow o_{qi})^2\Big)^2 \quad \text{(E.11)}$$

where η is a positive penalty parameter, N the number of output units and P the number of patterns. The derivative to (E.11) is

$$\frac{dE(\mathbf{w})}{do_{pj}} = \Leftrightarrow\frac{1}{NP}(t_{pj}\Leftrightarrow o_{pj})\Big(2 + 4\eta\frac{PN\Leftrightarrow 1}{PN}\big((t_{pj}\Leftrightarrow o_{pj})^2 \Leftrightarrow \tfrac{1}{PN}\textstyle\sum_i^N\sum_q^P(t_{qi}\Leftrightarrow o_{qi})^2\big) \quad \text{(E.12)}$$

Using the exponential error function shown in (E.9) and the *minimum variance* error function shown in (E.11) SCG and QP were again tested on the artificial data problem from section E.3.3. This time the algorithms were first terminated when all patterns were classified correctly or until a resonable limit was reached. Table E.3 and table E.4 summarizes the average results obtained. α was set to 1. The initial β was set to 0.9 and then halfed every time no errors were inside the penalized regions. The penalty parameter η was set to 1-2. In contrast to the runs with the least square error function both algorithms finds now in all runs optimal solutions with respect to correct classification. SCG has in average a speedup against QP of about 3.0. The exponential error function seems to yield the fastest convergence, but this might be because of the actual values of α, β and η.

E.5 Generalization

In this section we investigate the generalization ability of network solutions found by minimization of the different error functions. Again some artificial data was generated, this time with continuous input constrained between 0 and 1. We chose dimension 10 with 20 centerpoints, 50 distortions per centerpoint and 4 possible output classes. The average overlap between the centerpoints was 4%, meaning that 4% of the distortions were nearer other centerpoints than the one they were generated from. The set of patterns was then split in to a training set, validation set and a test set of equal size. When applying

Dim	SCG				QP				Speedup
	Epoch		Correct		Epoch		Correct		
	mean	std.dev.	mean	std.dev.	mean	std.dev.	mean	std.dev.	
8	82	13	1	0	265	68	1	0	1.6
10	147	23	1	0	758	226	1	0	2.6
12	111	10	1	0	840	184	1	0	3.8
14	81	5	1	0	414	90	1	0	2.6
16	94	13	1	0	524	78	1	0	2.8
18	89	4	1	0	470	36	1	0	2.6

Table E.4: Average results on artificial data using the minimum variance error function.

the k-nearest neighbor technique on the data we got a max performance of 94.26% on the validation set giving 93.69% on the test set (k=5). Because of the way the data is generated we would not expect the neural network solution to do much better than that. We ran the following experiments. QP was tested with and without primeoffset on the least square error function. SCG was tested on the least square error function, the exponential error function and the minimimum variance error function. 5 different runs were made for each test. When the classification rate of the validation set was at it highest the number of iterations run and the classification rate of the test set were recorded.

The results are illustrated in figure E.3. We observe the same trend for both the exponential error function and the minimum variance error function. The higher the α and η values the better the generalization. For η equal to 30 there is a decrease in generalization. At this point the constraint towards low variance was too strong. Unfortunatly, this gain in generalization is done at the expense of the convergence rate as the figure also show. This is, however, not surprising since high α and η values impose a tougher constraint on the acceptable path down to the minimum. The minimum variance- and the exponential error function gives approximately the same maximum generalization performance as the k-nearest neighbor. At this maximum generalization point the convergence rate of the minimum variance error funtion is slightly higher than the convergence rate of the exponential error function.

E.6 Conclusion

The conclusions to be made are twofold. First the paper has presented a comparison between two algorithms that both are known to be efficient. Empirically it has been shown that SCG has an average speedup against QP of about 3.0. Furthermore, SCG does not contain any problem-dependent parameters like QP's ϵ parameter. However, when the least square error function is used as objective function, SCG ends more often in suboptimal solutions with not as many correct classifications as QP. This is due to QP's ability to use a primeoffset term. By combining SCG and more suitable error functions for network training this problem is eliminated.

Second this paper has shown that imposing appropriate constraints on network solutions can improve convergence and generalization. We have proposed two new error functions that impose such constraints. We do not claim that these functions are in any way optimal, but we do believe that our results illustrates the neccesity of adding such

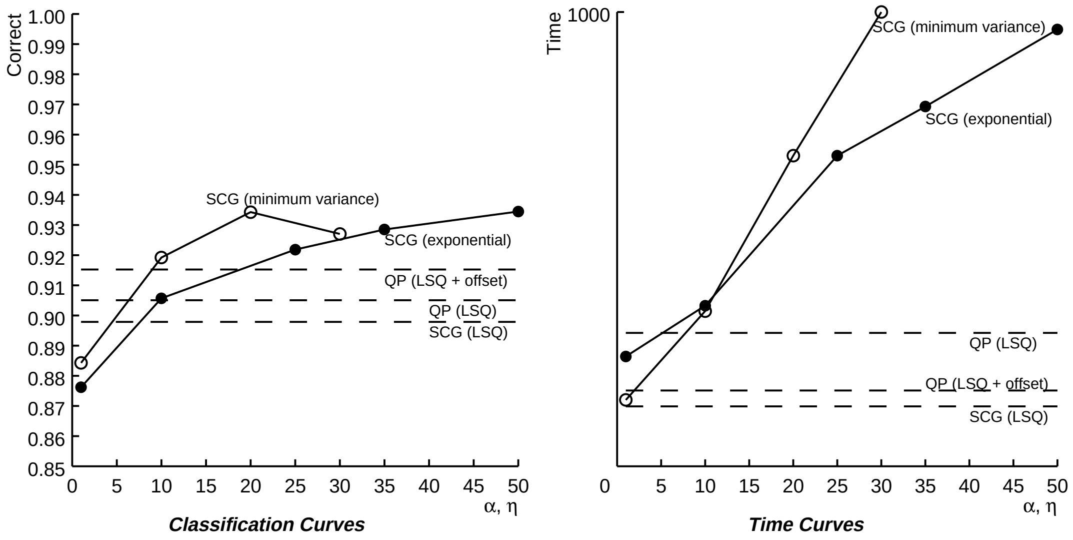

Figure E.3: Results on the test set using the exponential error function and the minimum variance error function with different α and η values.

constraints. Minimization with the new error functions produce in average better solutions with respect to generalization than the least square error function with the primeoffset added. SCG combined with these error functions yields faster convergence and better generalization than QP with primeoffset.

The quality of the solutions found with the new error functions depends heavily on the values of the constraint parameters α and η. We have not addressed the problem of choosing optimal values of α and η. Several heuristic methods could be applied, like starting with a small value and then slowly increase. More sophisticated techniques, like the ones used to estimate appropriate regularization parameters [Girard 89], might also be usuable in this context.

It would be interesting to know how the distribution of the errors on the training set influence the generalization ability. Our results indicate that the more balanced the distribution is, i.e, the more equal the errors are in magnitude, the better generalization one can expect. It remains to future work to actually prove the relationship between expected generalization and error distribution.

Acknowledgements

Many thanks to Wray Buntine for his helpful comments. Thanks also to John Hampshire for sharing some of his thesis results with us before publishing.

Bibliography

[Abramowitz 64] M. Abramowitz and I.A. Stegun, *Handbook of Mathematical Functions*, U.S. Department of Commerce, 1964.

[Akaike 59] H. Akaike (1959), *On a Successive Transformation of Probability Distribution and Its Application to the Analysis of the Optimum Gradient Method*, Ann. Inst. Statist. Math., Vol. 11, pp. 1-17.

[Aoki 71] M. Aoki (1971), *Introduction to Optimization Techniques*, The Macmillan Company, New York.

[Axelsson 77] O. Axelsson (1977), *Solution of Linear Systems of Equations: Iterative Methods*, In Sparse Matrix Techniques, Ed. V.A. Barker, Copenhagen, Lecture Notes in Mathematics 572, Springer Verlag, pp. 1-48.

[Axelsson 80] O. Axelsson (1980), *Conjugate Gradient Type Methods for Unsymmetric and Inconsistent Systems of Linear Equations*, Linear Algebra and its Applications, Vol. 29, Elsevier North Holland, inc., pp. 1-16.

[Battiti 89] R. Battiti (1989), *Accelerated Back-Propagation Learning: Two Optimization Methods*, Complex Systems, Vol. 3, pp. 331-342.

[Battiti and Masulli 90] R. Battiti and F. Masulli (1990), *BFGS Optimization for Faster and Automated Supervised Learning*, INCC 90 Paris, International Neural Network Conference, Vol. 2, pp. 757-760.

[Battiti 92] R. Battiti (1992), *First and Second-Order Methods for Learning: between Steepest descent and Newton's Method*, Neural Computation, Vol. 4 (2), pp. 141-167.

[Bishop 92] C. Bishop (1992), *Exact Calculation of the Hessian Matrix for the Multilayer Perceptron*, Neural Computation, Vol. 2, pp. 494-501.

[Brady and Raghavan 88] M. Brady and R. Raghavan (1988), *Gradient Descent Fails to Seperate*, Proceedings of the 1988 International Conference on Neural Networks, Vol. 1, pp. 649-656.

[Bryson and Ho 69] A.E. Bryson and Y.C. Ho (1969), *Applied Optimal Control*, New York: Blaisdell.

[Buntine and Weigend 91a] W. Buntine and A. Weigend (1991), *Calculating Second Derivatives on Feed-Forward Networks*, submitted to IEEE Transactions on Neural Networks.

[Buntine and Weigend 91b] W.L. Buntine and A.S. Weigend (1991), *Bayesian Back-Propagation*, Complex Systems, Vol. 5, pp. 603-643.

[Cater 87] J.P. Cater (1987), *Successfully Using Peak Learning Rates of 10 (and Greater) in Back-Propagation Networks with the Heuristic Learning Algorithm*, In IEEE First International Conference on Neural Networks, San Diego, Eds. M. Caudill and C. Butler, Vol. 2, pp. 645-651.

[Cauchy 1847] A. Cauchy (1847), *Méthode Général pour la Résolution des Systéms d'Équations Simulationées*, Comp. rend. Acad. Sci. Paris, pp. 536-538.

[Chan and Fallside 87] L.W. Chan and F. Fallside (1987), *An Adaptive Training Algorithm for Back-Propagation Networks*, Computer Speech and Language, Vol. 2, pp. 205-218.

[Chan 90] L.W. Chan (1990), *Efficacy of Different Learning Algorithms of Back-Propagation Networks*, In Proceedings IEEE TENCON-90.

[Chung 54] K. Chung (1954), *On a Stochastic Approximation Method*, Ann. Math. Stat., Vol. 25, pp. 463-483.

[Cochran 77] W.G. Cochran (1977), *Sampling Techniques*, John Wiley & Sons, Inc.

[Concus et al. 76] P. Concus, G.H. Golub and D.P. O'Leary (1976), *A Generalized Conjugate Gradient Method for the Numerical Solution op Elliptic Partial Differential Equations*, In Sparse Matrix Computations, Ed. J.R Bunch and D.J. Rose, Academic Press, New York, pp. 309-332.

[Darken et al. 92] C. Darken, J. Chang and J. Moody (1992), *Learning Rate Schedules for Faster Stochastic Gradient Search*, In Neural networks for Signal Processing 2, IEEE Workshop, Eds. S.Y Kung, F. Fallside, J. Å. SØrensen and C.A. Kamm, IEEE Press., pp. 3-13.

[Darken 93] C. Darken (1993), Personal communication.

[Dixon and Price 89] L.C.W. Dixon and R.C. Price (1989), *Truncated Newton Method for Sparse Unconstrained Optimization Using Automatic Differentiation*, Journal of Optimization Theory and Applications, Vol. 60, No. 2, pp. 261-275.

[Fahlman 89] S.E. Fahlman (1989). *Fast Learning Variations on Back-propagation: An Empirical Study*, In proceedings of the 1988 Connectionist Models Summer School, Eds. D.S. Touretzky, G. Hinton and T. Sejnowski, pp. 38-51, San Mateo: Morgan Kauffmann.

[Fedorov 72] V.V. Federov (1972), *Theory of Optimal Experiments*, Academic Press, New York.

[Fletcher 75] R. Fletcher (1975). *Practical Methods of Optimization*, Vol. 1, John Wiley & Sons.

[Franzini 87] M.A. Franzini (1987). *Speech Recognition with Back-Propagation*, In Proceedings of the Ninth Annual Conference of the IEEE Engineering in Medicine and Biology Society, Boston, pp. 1702-1703.

[Gallager 68] R.G. Gallager (1968), *Information Theory and Reliable Communication*, John Wiley & Sons, Inc.

[Gill and Murray 74] P.E. Gill and W. Murray (1974), *Safeguarded Steplength Algorithms For Optimization Using Descent Methods*, National Physica Laboratory, Division of Numerical Analysis and Computing, NPL Report NAC 37.

[Gill et al. 81] P.E. Gill, W. Murray and M.H. Wright (1981). *Practical Optimization*, Academic Press Inc., London.

[Girard 89] D.A. Girard (1989), *A Fast 'Monte-Carlo Cross-Validation' Procedure for Large Lesat Squares Problems with Noisy Data*, Numer. Math., Vol. 56, pp. 1-23.

[Gish 90] H. Gish (1990), *A Probabilistic Approach to the Understanding and Training of Neural Network Classifiers*, In Proceedings of the 1990 IEEE International Conference on Acoustics, Speech and Signal Processing, Vol. 3, pp. 1361-1364.

[Goldstein 87] L. Goldstein (1987), *Mean Square Optimality in the Continuous Time Robbins Monro Procedue*, Technical Report DRB-306, Department of Mathematics, University of Southern California.

[Golub and Loan] G.H. Golub and C.F. van Loan (1983), *Matrix Computations*, The John Hopkins University Press,

[Haffner et al. 88] P. Haffner, A. Waibel, H. Sawai and K. Shikano (1988), *Fast Back-Propagation Learning Methods for Neural Networks in Speech*, ATR Interpreting Telephony Research Laboratories.

[Hampshire 92] J.B. Hampshire (1992), *A Differential Theory of Learning for Statistical Pattern Recognition with Connectionist Models*, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University.

[Hampshire and Waibel 90] J.B. Hampshire and A.H. Waibel (1990), *A Novel Objective Function for Improved Phoneme Recognition Using Time-Delay Neural Networks*, IEEE Transactions on Neural Networks, Vol. 1, No. 2, pp. 216-228.

[Hassibi and Stork 93] B. Hassibi and D.G. Stork (1993), *Second Order Derivatives for Network Pruning: Optimal Brain Surgeon*, In Neural Information Processing Systems, Ed. Cowan and Giles, Morgan Kaufmann, Vol. 4.

[Hastie and Tibshirani 90] T.J. Hastie and R.J. Tibshirani (1990), *Generalised Additive Models*, London, Chapman and Hall.

[Hestenes and Stiefel 52] M.R. Hestenes and S. Stiefel (1952), *methods of Conjugate Gradient for Solving Linear Systems*, J. Res. Nat. Bur. Standards, Vol. 49, pp. 409-436.

[Hinton 89] G. Hinton (1989), *Connectionist Learning Procedures*, Artificial Intelligence, Vol. 40, pp. 185-234.

[Horn and Johnson 85] R.H. Horn and C.A. Johnson (1985), *Matrix Analysis*, Cambridge University Press, Cambridge.

[Jacobs 88] R.A. Jacobs (1988), *Increased Rates of Convergence Through Learning Rate Adaptation*, Neural Networks, Vol. 1, pp. 295-307.

[Johansson et al. 91] E.M. Johansson, F.U. Dowla and D.M. Goodman (1991), *Backpropagation Learning for Multi-Layer Feed-Forward Neural Networks Using the Conjugate Gradient Method*, International Journal of Neural Systems, Vol. 2, No. 4, pp. 291-301.

[Judd 87] J.S. Judd (1987),*Complexity of connectionist learning with various node functions*, COINS Technical Report 87-60, University of Amherst, Amherst, MA.

[Kailath 80] T. Kailath (1980), Linear Systems, Prentice Hall.

[Karle 91] J. Karle, *Direct calculation of atomic coordinates from diffraction intensities: Space group P1*, Proceedings of the National Academy of Sciences, USA, Vol. 1, pp. 10099-10103.

[Kinsella 92] J.A. Kinsella (1992), *Comparison and Evaluation of Variants of the Conjugate Gradient Method for Efficient Learning in Feed-Forward Neural Networks with Backward Error Propagation*, Network, Vol. 3, pp. 27-35.

[Knuth 81] D.E. Knuth (1981), *The Art of Computer Programming*, Vol. 2, Semi-Numerical Algorithms, Addison-Wesley Publishing Company.

[Kramer et al. 88] A.H. Kramer and A. Sangiovanni-Vicentelli (1988), *Efficient Parallel Learning Algorithms for Neural Networks*, In Advances in Neural Information Processing Systsems, Morgan Kaufmann, San Mateo, Vol. 1, pp. 75-89.

[Kreyszig 88] E. Kreyszig (1988), *Advanced Engineering Mathematics*, 6th edition, John Wiley and Sons, Inc.

[Kuhn and Herzberg 90] G.M. Kuhn and P. Herzberg (1990), *Some Variations on Training of Recurrent Networks*, In Proceedings of CAIP Neural Networks Workshop, Rutgers University, pp. 15-17.

[Kuhn and Watrous 93] G.M. Kuhn and R.L. Watrous (1993), *Comparison of Feed-forward and Recurrent Sensivities in Speech Recognition*, in Artificial Neural Networks with Applications in Speech and Vision, Ed. R. Mammone, London, Chapman & Hall.

[Lang and Witbrock 89] K.J. Lang and M. Witbrock (1989), *Learning to Tell Two Spirals Apart*, In proceedings of the 1988 Connectionist Models Summer School, Eds. D.S. Touretzky, G. Hinton and T. Sejnowski, pp. 52-59, San Mateo: Morgan Kauffmann.

[Le Cun 89] Y. Le Cun (1989). *Generalization and Network Design Strategies*, In Connectionism in Perspective, Eds. R. Pfeifer, Z. Schleter, F. Fogelmann and L. Steels, Zurich, Elsevier.

[Le Cun et al. 92] Y. Le Cun, J.S. Denker and S.A. Solla (1990), *Optimal Brain Damage*, In Neural Information Processing Systems, Ed. D.S Touretzky, Morgan Kaufmann, Vol. 2., pp.598-605.

[Le Cun et al. 91] Y. Le Cun, I. Kanter, S. Solla (1991), *Eigenvalues of Covariance Matrices: Application to Neural Network Learning*, Physical Review Letters, Vol. 66, pp. 2396-2399.

[Le Cun et al. 93] Y. Le Cun, P.Y. Simard and B. Pearlmutter (1993), *Automatic Learning Rate Maximization by On-line Estimation of the Hessian's Eigenvectors*, in Proceedings of Neural Information Processing Systems, Vol. 5, Eds. Giles, Hanson and Cowan, Morgan Kauffman.

[Luenberger 84] D.G. Luenberger (1984), *Linear and Nonlinear Programming*, Addison-Wesley Publishing Company, Inc.

[MacKay 91a] D.J.C. MacKay (1991), *Bayesian Interpolation*, Neural Computation, Vol. 4, N0. 3, pp. 415-447.

[MacKay 91b] D.J.C. MacKay (1991), *A Practical Bayesian Framework for Back-Prop Networks*, Neural Computation, Vol. 4, N0. 3, pp. 448-472.

[MacKay 92] D.J.C. MacKay (1992), *Information-Based Objective Functions for Active Data Selection*, Neural Computation, Vol. 4, pp. 590-604.

[Makram-Ebeid et al. 89] S. Mackram-Ebeid, J.A. Surat and J. Viola (1989), *A Rationalized Backpropagation Learning Algorithm*, In proceedings of the International Joint Conference on Neural Networks, Washington 1989, Vol. 2, pp. 373-380, New York: IEEE.

[Mingers 89] J. Mingers (1989), *An Empirical Comparison of Selection Measures for Decision-Tree Induction*, Machine Learning, Vol. 3, pp. 319-342.

[Moody 92] J.E. Moody (1992), *The effective number of parameters: an analysis of generalization and regularization in nonlinear learning systems*, In Neural Information Processing Systems, Ed. Cowan and Giles, Morgan Kaufmann, Vol. 4.

[Møller 90a] M. Møller (1990), *CM Algoritmen*, Masters Thesis, Daimi IR-95, Computer Science Department, Aarhus University.

[Møller 90b] M. Møller (1990), *Learning by Conjugate Gradients*, In Proceedings of the Sixth International Meeting of Young Computer Scientist, LNCS 464, Springer Verlag, New York, pp. 184-195.

[Møller 93a] M. Møller (1993), *A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning*, Neural Networks, June, Vol. 6, No. 4, pp. 525-533.

[Møller 93b] M. Møller (1993), *Supervised Learning on Large Redundant Training sets*, International Journal of Neural Systems, Vol. 4, No. 1, pp. 15-25.

[Møller and Fahlman 93] M. Møller and S.E. Fahlman (1993), *Supervised Learning: Improving Network Solutions*, in preparation.

[Møller 93c] M. Møller (1993), *Exact Calculation of the Product of the Hessian Matrix of Feed-Forward Network Error Functions and a Vector in $O(N)$ Time*, Technical Report, Daimi PB-432, Computer Science Department, Aarhus University.

[Møller 93d] M. Møller (1993), *Adaptive Preconditioning of the Hessian Matrix*, submitted to Neural Computation.

[Orfanidis 90] S.J. Orfanidis (1990), *Gram-Schmidt Neural Nets*, Neural Computation, Vol. 2, pp. 116-126.

[Parker 85] D.B. Parker (1985), *Learning Logic*, Technical Report TR-47, Center for Computational Research in Economics and Management Science, Massachusetts Institute of Technology, Cambridge, MA.

[Pearlmutter 93] B.A. Pearlmutter (1993), *Fast Exact Multiplication by the Hessian*, preprint, to appear in Neural Computation.

[Plaut et al. 86] D. Plaut, S. Nowlan and G. Hinton (1986), *Experiments on Learning by Back-Propagation*, Technical Report CMU-CS-86-126, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[Plutowski et al.] M. Plutowski, G. Cottrell and H. White, *Learning Mackey-Glass from 25 examples, Plus or Minus 2*, In Proceedings of Neural Information Processing Systems, Vol. 4, Eds. Giles, Hanson and Cowan, Morgan Kauffman.

[Powell 77] M. Powell (1977), *Restart Procedures for the Conjugate Gradient Method*, Mathematical Programming, pp. 241-254.

[Press et al. 88] W.H. Press, B.P. Flannery, S.A. Teucolsky and W.T. Verrerling (1988), *Numerical Recipes in C*, Cambridge University Press.

[Ralston et al. 78] A. Ralston and P. Rabinowitz (1978), *A First Course in Numerical Analysis*, McGraw-Hill Book Company, Inc.

[Rissanen 84] J. Rissanen (1984), *Universal Coding, Information , Prediction, and Estimation*, IEEE Transactions on Information Theory, Vol. 30, No. 4, pp. 629-636.

[Robbins and Monro 51] H. Robbins and S. Munro (1951), *A Stochastic Approximation Method*, Ann. Math. Stat., Vol. 22, pp. 400-407.

[Rumelhart et al. 86] D.E. Rumelhart, G.E. Hinton and R.J. Williams (1986), *Learning Internal Representations by Error Propagation*, In Parallel Distributed Processing, Nature 323, pp. 533-536.

[Schaffer 92] C. Schaffer (1992), *Sparse Data and the Effect of Overfitting Avoidance in Decision Tree Induction*, In proceedings of AAAI-92.

[Seber and Wild 89] G.A.F. Seber and C.J Wild (1989), *Nonlinear Regression*, John Wiley and Sons, New York.

[Sejnowski and Rosenberg 87] T.J. Sejnowski and C.R. Rosenberg (1987), *Parallel networks that learns to pronounce English text*, Complex Systems, Vol. 1, pp. 145-168.

[Shannon and Warren 64] C.E. Shannon and W. Warren (1964), *The Mathematical Theory of Communication*, The university of Illinois Press, Urbana.

[Silva and Almeida 90] F. Silva and L. Almeida (1990), *Acceleration Techniques for the Back-Propagation Algorithm*, Lecture Notes in Computer Science, Springer Verlag, Vol. 412, pp. 110-119.

[Skilling 89] J. Skilling (1989), *The Eigenvalues of Mega-Dimensional Matrices*, In Maximum Entropy and Bayesian Methods, Editor J. Skilling, Kluwer Academic Publishers, pp. 455-466.

[Sluis and Horst 86] A. van der Sluis and H.A. van der Horst (1986), *The Rate of Convergence of Conjugate Gradient*, Numer. Math., Vol. 48, pp. 543-560.

[Solla et al. 88] S.A. Solla, E. Levin and M. Fleisher (1988), *Accelerated Learning in Layered Neural Networks*, Complex Systems, Vol. 2, pp. 625-639.

[Tesauro 87] G. Tesauro (1987), *Scaling relationships in back-propagation learning: Dependence on training set size.* Complex Systems, Vol. 2, pp. 367-372.

[Tollenaere 90] T. Tollenaere (1990), *SuperSAB: Fast Adaptive Back-Propagation with Good Scaling Properties*, Neural Networks, Vol. 3, pp. 561-573.

[Vogl et al. 88] T.P. Vogl, J.K. Mangis, A.K. Rigler, W.T. Zink and D.L. Alkon (1988), *Accelerating the Convergence of the Back-Propagation Method*, Biological Cybernetics, Vol. 59, pp. 257-263.

[Wan 90] E.A. Wan (1990), *Neural Network Classification: A Bayesian Interpretation*, IEEE Transactions on Neural Networks, Vol. 1 (4), pp. 303-305.

[Watrous 87] R.L. Watrous (1987), *Learning Algorithms for Connectionist Networks: Applied Gradient Methods of Nonlinear Optimization*, In IEEE First International Conference on Neural Networks, San Diego, Eds. M. Caudill and C. Butler, Vol. 2, pp. 619-627.

[Weigend et al. 90] A.S. Weigend, B.A. Huberman and D.E. Rumelhart (1990), *Predicting the Future: A Connectionist Approach*, International Journal of Neural Systems, Vol. 1, pp.193-209.

[Werbos 74] P. Werbos (1974), *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Ph.D. Thesis, Harvard University.

[Widrow and Stearns 85] B. Widrow and S.D. Stearns (1985), *Adaptive Signal Processing*, Prentice Hall, Englewood Cliffs, NJ.

[Williams 91] P.M. Williams (1992), *A Marquardt Algorithm for Choosing the Step-size in Back-Propagation Learning with Conjugate Gradients*, Technical Report CSRP-229, Cognitive Science, University of Sussex.

[Yang 92] U.M. Yang (1992), *Preconditioned Conjugate Gradient-Like Methods for Nonsymmetric Linear Systems*, CSRD Report 1210, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign.

[Yoshida 91] T. Yoshida (1991), *A Learning Algorithm for Multilayered Neural Networks: A Newton Method Using Automatic Differentiation*, In Proceedings of International Joint Conference on Neural Networks, Seattle, Poster.

[Yu and Simmons 90] Y. Yu and R.F. Simmons (1990). *Descending Epsilon in Back-Propagation: A Technique for Better Generalization*, In proceedings of the International Joint Conference on Neural Networks, Washington 1990, Vol. 3, pp. 167-172, New York: IEEE.