Exact Calculation of the Product of the Hessian Matrix of Feed-Forward Network Error Functions and a Vector in $O(N)$ Time

Martin Møller[*]
Computer Science Department
Aarhus University
DK-8000 Århus, Denmark

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 $O(N)$ time, where 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.

1 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

[*]This work was done while the author was visiting School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[Møller 93a], and in recent network pruning techniques [MacKay 91], [Hassibi 92]. Several researchers have recently derived formulae for exact calculation of the elements in the Hessian matrix [Buntine 91], [Bishop 92]. In the general case exact calculation of the Hessian matrix needs $O(N^2)$ time- and 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 91]. 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 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 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.

2 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*.

$$\begin{aligned} S_m^l &= \left\{ n_s^r \in \aleph \mid \text{ There is a connection from } n_s^r \text{ to } n_m^l, \;\; r < l, \;\; 1 \leq s \leq N_r \right\} \quad (1) \\ T_m^l &= \left\{ n_s^r \in \aleph \mid \text{ There is a connection from } n_m^l \text{ to } n_s^r, \;\; r > l, \;\; 1 \leq s \leq N_r \right\} \end{aligned}$$

The training set accociated with network $\aleph$ is

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

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

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

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

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

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

$$\frac{\partial v_{pi}^h}{\partial v_{pm}^l} = \sum_{n_s^r \in T_m^l} \frac{\partial v_{pi}^h}{\partial v_{ps}^r} \frac{\partial v_{ps}^r}{\partial v_{pm}^l} = f'(v_{pm}^l) \sum_{n_s^r \in T_m^l} w_{sm}^{rl} \frac{\partial v_{pi}^h}{\partial v_{ps}^r} \tag{5}$$

$$\frac{\partial E_p}{\partial v_{pm}^l} = \sum_{n_s^r \in T_m^l} \frac{\partial E_p}{\partial v_{ps}^r} \frac{\partial v_{ps}^r}{\partial v_{pm}^l} = f'(v_{pm}^l) \sum_{n_s^r \in T_m^l} w_{sm}^{rl} \frac{\partial E_p}{\partial v_s^r} \tag{6}$$

3 Calculation of the Hessian matrix 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 $\mathbf{w}$.

$$\begin{aligned} H_p(\mathbf{w})\mathbf{d} &= \frac{d}{d\mathbf{w}}(\mathbf{d}^T\frac{dE_p}{d\mathbf{w}}) = \frac{d}{d\mathbf{w}}(\mathbf{d}^T\sum_{j=1}^{N_L}\frac{\partial E_p}{\partial v_{pj}^L}\frac{dv_{pj}^L}{d\mathbf{w}}) \qquad (7)\\ &= \sum_{j=1}^{N_L}\frac{\partial^2 E_p}{(\partial v_{pj}^L)^2}(\mathbf{d}^T\frac{dv_{pj}^L}{d\mathbf{w}})\frac{dv_{pj}^L}{d\mathbf{w}} + \frac{\partial E_p}{\partial v_{pj}^L}(\frac{d^2 v_{pj}^L}{d\mathbf{w}^2}\mathbf{d})\\ &= \sum_{j=1}^{N_L}\Big(f'(v_{pj}^L)^2\frac{\partial^2 E_p}{(\partial u_{pj}^L)^2} + f''(v_{pj}^L)\frac{\partial E_p}{\partial u_{pj}^L}\Big)(\mathbf{d}^T\frac{dv_{pj}^L}{d\mathbf{w}})\frac{dv_{pj}^L}{d\mathbf{w}} + \frac{\partial E_p}{\partial v_{pj}^L}(\frac{d^2 v_{pj}^L}{d\mathbf{w}^2}\mathbf{d}), \end{aligned}$$

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

$$\mathbf{A} = \sum_{j=1}^{N_L}\Big(f'(v_{pj}^L)^2\frac{\partial^2 E_p}{(\partial u_{pj}^L)^2}+f''(v_{pj}^L)\frac{\partial E_p}{\partial u_{pj}^L}\Big)(\mathbf{d}^T\frac{dv_{pj}^L}{d\mathbf{w}})\frac{dv_{pj}^L}{d\mathbf{w}} \quad \text{and} \quad \mathbf{B} = \sum_{j=1}^{N_L}\frac{\partial E_p}{\partial v_{pj}^L}(\frac{d^2 v_{pj}^L}{d\mathbf{w}^2}\mathbf{d}). \qquad (8)$$

We first concentrate on calculating the **A**-vector.

Lemma 1 *Let φ_{pm}^l be defined as $\varphi_{pm}^l = \mathbf{d}^T\frac{dv_{pm}^l}{d\mathbf{w}}$. φ_{pm}^l can be calculated by forward propagation using the recursive formula*

$\varphi_{pm}^l = \sum_{n_s^r\in S_m^l}\Big(d_{ms}^{lr}u_{ps}^r + w_{ms}^{lr}f'(v_{ps}^r)\varphi_{ps}^r\Big) + d_m^l \quad ,l>0 \ ,\qquad \varphi_{pi}^0 = 0 \ ,1\le i\le N_0$.

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

$$\begin{aligned} \varphi_{pm}^l &= \mathbf{d}^T\frac{dv_{pm}^l}{d\mathbf{w}} = \mathbf{d}^T(\sum_{n_s^r\in S_m^l}\frac{d}{d\mathbf{w}}(w_{ms}^{lr}u_{ps}^r) + \frac{dw_m^l}{d\mathbf{w}})\\ &= \sum_{n_s^r\in S_m^l}\Big(w_{ms}^{lr}f'(v_{ps}^r)\mathbf{d}^T\frac{dv_{ps}^r}{d\mathbf{w}} + d_{ms}^{lr}u_{ps}^r\Big) + d_m^l = \sum_{n_s^r\in S_m^l}\Big(d_{ms}^{lr}u_{ps}^r + w_{ms}^{lr}f'(v_{ps}^r)\varphi_{ps}^r\Big) + d_m^l \end{aligned}$$

$\square$

Lemma 2 *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

$$\mu^l_{pm} = f'(v^l_{pm}) \textstyle\sum_{n^r_s \in T^l_m} w^{rl}_{sm} \mu^r_{ps} \quad , l < L \ ,$$
$$\mu^L_{pj} = \Big(f'(v^L_{pj})^2 \tfrac{\partial^2 E_p}{(\partial u^L_{pj})^2} + f''(v^L_{pj}) \tfrac{\partial E_p}{\partial u^L_{pj}}\Big) \varphi^L_{pj} \quad , 1 \le j \le N_L.$$

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 $k > l$.

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

$\square$

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

Lemma 3 *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

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

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

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

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. $\square$

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 2 and lemma 3.

Proof. By combination of lemma 2 and lemma 3. $\square$

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

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

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

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

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

Proof. By corollary 1 and equation 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: $\varphi_{pi}^0 = 0$.

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

   $$v_{pm}^l = \textstyle\sum_{n_s^r \in S_m^l} w_{ms}^{lr} u_{ps}^r + w_m^l \ , \qquad u_{pm}^l = f(v_{pm}^l),$$
   $$\varphi_{pm}^l = \textstyle\sum_{n_s^r \in S_m^l} \left(d_{ms}^{lr} u_{ps}^r + w_{ms}^{lr} f'(v_{ps}^r)\varphi_{ps}^r \right) + d_m^l.$$

3. Output layer.

   For nodes $j = 1$ to N_L do

   $$\delta_{pj}^L = \frac{\partial E_p}{\partial v_{pj}^L} \ , \quad \beta_{pj}^L = 0 \ , \quad \mu_{pj}^L = \left(f'(v_{pj}^L)^2 \frac{\partial^2 E_p}{(\partial u_{pj}^L)^2} + f''(v_{pj}^L) \frac{\partial E_p}{\partial u_{pj}^L} \right) \varphi_{pj}^L.$$

   For all nodes $n_s^r \in S_j^L$ do

   $$\mathbf{Hd}_{js}^{Lr} = \mathbf{Hd}_{js}^{Lr} + \delta_{pj}^L f'(v_{ps}^r)\varphi_{ps}^r + \mu_{pj}^L u_{ps}^r \ , \quad \mathbf{Hd}_j^L = \mathbf{Hd}_j^L + \mu_{pj}^L \ ,$$
   $$\mathbf{G}_{js}^{Lr} = \mathbf{G}_{js}^{Lr} + \delta_{pj}^L u_{ps}^r \ , \quad \mathbf{G}_j^L = \mathbf{G}_j^L + \delta_{pj}^L.$$

4. Backward propagation.

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

   $$\mu_{pm}^l = f'(v_{pm}^l) \textstyle\sum_{n_s^r \in T_m^l} w_{sm}^{rl} \mu_{ps}^r \ , \quad \delta_{pm}^l = f'(v_{pm}^l) \textstyle\sum_{n_s^r \in T_m^l} w_{sm}^{rl} \delta_{ps}^r,$$
   $$\beta_{pm}^l = \textstyle\sum_{n_s^r \in T_m^l} \left(f'(v_{pm}^l) w_{sm}^{rl} \beta_{ps}^r + \left(d_{sm}^{rl} f'(v_{pm}^l) + w_{sm}^{rl} f''(v_{pm}^l)\varphi_{pm}^l \right)\delta_{ps}^r \right).$$

   For all nodes $n_s^r \in S_m^l$ do

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

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

4 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.

4.1 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

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

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

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

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

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

where $\mathbf{s}_t$ is

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

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

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

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

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

$\mathbf{s}_t$ 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 (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 (15) can be as high as $\frac{2\varepsilon}{\sigma_t}$ [Ralston 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].

4.2 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 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 85]. This implies that the inverse of the

Hessian matrix $H(\mathbf{w}_t)^{-1}$ can be written in the form

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

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

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

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

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

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 (18) reduces the component of the gradient along the directions with large curvature. See also [Le Cun 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 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 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:

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

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

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 (15). Now this term can be calculated exactly by use of the algorithm described in the last sections.

5 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 $O(N)$ time- and memory requirements, where N is the number of variables in the network.

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.

References

[Bishop 92] C. Bishop (1992), *Exact Calculation of the Hessian Matrix for the Multilayer Perceptron*, Neural Computation, Vol. 2, pp. 494-501.

[Buntine 91] W. Buntine and A. Weigend (1991), *Calculating Second Derivatives on Feed-Forward Networks*, submitted to IEEE Transactions on Neural Networks.

[Le Cun 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 93] Y. Le Cun, P.Y. Simard and B. Pearlmutter (1993), *Local Computation of the Second Derivative Information in a Multilayer Network*, in Proceedings of Neural Information Processing Systems, Morgan Kauffman, in print.

[Dixon 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.

[Fletcher 75] R. Fletcher (1975). *Practical Methods of Optimization*, Vol. 1, John Wiley & Sons.

[Hassibi 92] B. Hassibi and D.G. Stork (1992), *Second Order Derivatives for Network Pruning: Optimal Brain Surgeon*, In Proceedings of Neural Information Processing Systems, Morgan Kauffman.

[Horn 85]	R.H. Horn and C.A. Johnson (1985), *Matrix Analysis*, Cambridge University Press, Cambridge.
[MacKay 91]	D.J.C. MacKay (1991), *A Practical Bayesian Framework for Back-Prop Networks*, Neural Computation, Vol. 4, N0. 3, pp. 448-472.
[Møller 93a]	M. Møller (1993), *A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning*, Neural Networks, in press.
[Møller 93b]	M. Møller (1993), *Supervised Learning on Large Redundant Training sets*, International Journal of Neural Systems, in press.
[Pearlmutter 93]	B.A. Pearlmutter (1993), *Fast Exact Multiplication by the Hessian*, preprint, submitted.
[Ralston 78]	A. Ralston and P. Rabinowitz (1978), *A First Course in Numerical Analysis*, McGraw-Hill Book Company, Inc.
[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.