Second Order Backpropagation — Efficient Computation of the Hessian Matrix for Neural Networks

Raúl Rojas *

TR-93-057

28 September 1993

Abstract

Traditional learning methods for neural networks use some kind of gradient descent in order to determine the network's weights for a given task. Some second order learning algorithms deal with a quadratic approximation of the error function determined from the calculation of the Hessian matrix, and achieve improved convergence rates in many cases. We introduce in this paper *second order backpropagation*, a method to calculate efficiently the Hessian of a linear network of one-dimensional functions. This technique can be used to get explicit symbolic expressions or numerical approximations of the Hessian and could be used in parallel computers to improve second order learning algorithms for neural networks. It can be of interest also for computer algebra systems.

*Freie Universität Berlin, Takustr. 9, Berlin 14195

1 Introduction

Traditional learning algorithms for feed-forward neural networks use gradient descent techniques to find a local minimum of the error function and the corresponding network's weights. Several authors have proposed going further than the standard first-order methods by using some kind of second-order approximation to the error function and appropriate learning algorithms [Battiti 92]. Newton's method, in particular, uses this kind of approach. For these and some other reasons, methods for the determination of the Hessian matrix in the case of multilayered networks have been studied recently [Bishop 92].

We show in this paper how to compute efficiently the elements of the Hessian matrix using a graphical approach which reduces the whole problem to a computation by inspection. Our method is more general than Bishop's because arbitrary topologies can be handled. The only restriction that we impose on the network is that it should contain no cycles, i.e. it should be of the feed-forward type. We first show how backpropagation itself can be visualized as a graph labeling algorithm, and how second order partial derivatives can be introduced. The method should be of interest for researchers who do not want to derive analytically the Hessian matrix each time the network topology changes and also for practitioners looking for efficient algorithms for the computation of the Hessian.

This paper is organized as follows: in the second section we give some definitions and look at the backpropagation algorithm introducing our graphical approach for first-order methods. In the third section we handle the case of second order derivatives in networks of functions. The next section discusses some examples and the general case for the computation of the Hessian. We conclude with a discussion of these results.

2 First order backpropagation

The kind of neural networks we consider are directed graphs without cycles. Information is fed into the network through selected input nodes and flows through the weighted edges of the graph. All other nodes in the network collect incoming information additively and compute a one-dimensional function of this input giving the result off. The result of the computation is produced at the output nodes and represents a function of the input which we call the *network function*.

In the learning phase, backpropagation networks are extended by including new nodes connected to the output nodes in which the current output is compared with the expected target value and the square of the difference is computed. The sum of all differences at the additional nodes is collected at another node and the result is the error function for a single input pattern (Figure 1). We call it an *error function* because it depends on the weights selected for the edges of the directed graph. In a network with n different weights $w_1, w_2, \ldots, w_n$ the error function for the i-th pattern will be denoted by E_i. The total error function E for all p patterns presented to the network is just $E = E_1 + E_2 + \cdots + E_p$. The backpropagation algorithm performs gradient descent on the error function trying to find the combination of weights which minimizes the mean squared error for all patterns. It is thus necessary

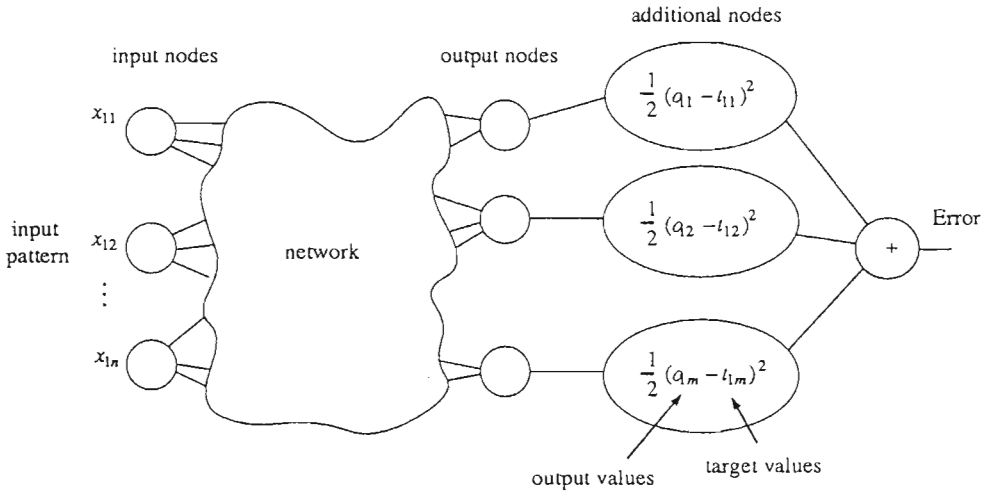

Figure 1: Extended neural network

to compute the gradient of the error function

$$\nabla E = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_n}).$$

The weight corrections are performed according to the updating rule

$$\Delta w_i = -\gamma \frac{\partial E}{\partial w_i} = -\gamma(\frac{\partial E_1}{\partial w_i} + \frac{\partial E_2}{\partial w_i} + \cdots + \frac{\partial E_p}{\partial w_i}),$$

where γ is a learning parameter. We are thus lead to consider the following problem: given a neural network and its i-th input, how do we derive the partial derivatives of E_i for $i = 1, \ldots, p$ according to each weight in the network?.

Figure 2 shows the most simple case: a composition of two one dimensional functions f and g. The input to this network is x and we look for the partial derivative of the network function $f(w_2 g(w_1 x))$ with respect to w_1. This is of course $f'(w_2 g(w_1 x)) w_2 g'(w_1 x) x$. This result can be computed graphically from Figure 2 by thinking of each node in the network as divided in two parts: the right side stores the result of the feed-forward computation; the left side stores the derivative of the node's function evaluated at the received input. The backpropagation algorithm amounts just to going backwards *multiplicatively* collecting all the left sides of the nodes and transmitting the result through the weighted edges of the network. This is backpropagation in a nutshell: a node labeling algorithm to efficiently implement the chain rule of differential calculus. It is evident that the above procedure can be generalized by induction to any number of nodes connected in sequence.

The only additional case that we have to consider is when there exist several paths from the output of the network to the weight being considered. Figure 3 shows such a network. The network function being computed is $F(x) = f_1(wx) + f_2(wx)$. The partial derivative of F with respect to w is evidently

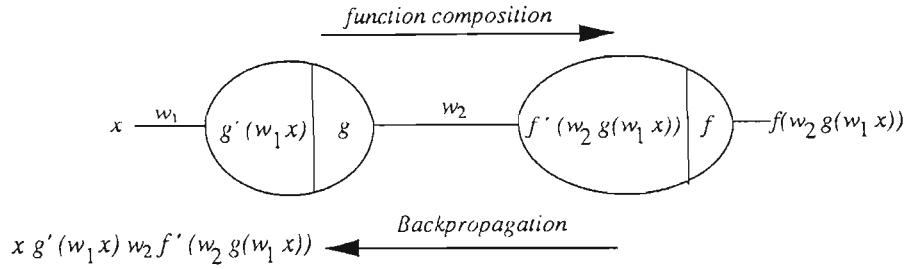

Figure 2: Function composition

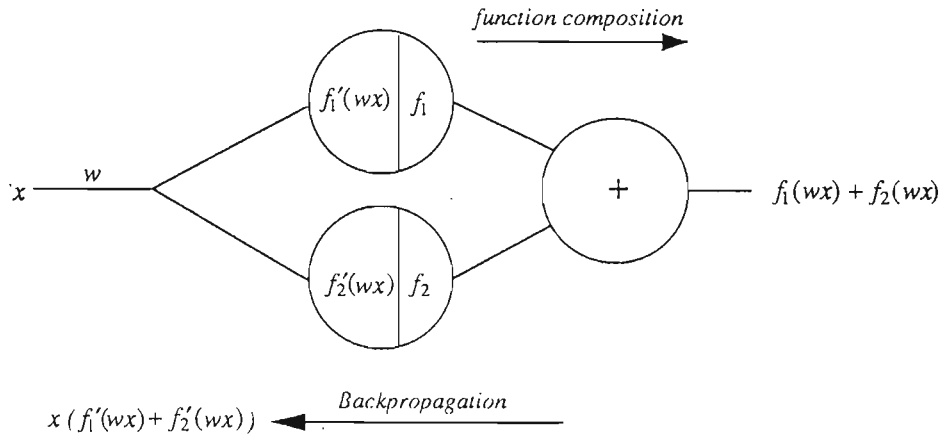

Figure 3: Addition of functions

$$\frac{\partial F}{\partial w} = x(f_1'(wx) + f_2'(wx)).$$

In this case the backpropagation algorithm just prescribes going from the output of the network backwards, collecting multiplicatively all left sides of the neurons found on each possible path up to the weight being considered. At the points where alternative paths meet again the partial results are added up. Note that to implement this algorithm only the reversibility of the network is assumed, that is absence of cycles. It is very easy to show [Rojas 93a] that the two rules given above lead to the same expressions for the backpropagation algorithm in multilayer networks derived with so much pain in some books. It is only necessary to inspect the network and collect the necessary terms. The proof by induction that the method works with any kind of feed-forward network is easy to perform and can be found in [Rojas 93b].

We need a little bit of additional notation in order to describe first and second

order backpropagation. We will label the weights in a network consecutively, just like $w_1, w_2, \ldots, w_m$. The nodes of the network will be labeled in the same manner. But we will refer also to weight w_{ij}, meaning the weight associated with the edge between node i and node j. Both forms of labeling will be used concurrently. This helps to avoid notational clutter.

Note that in a feed-forward network for every input pattern not only an output node produces a result F which is a function of the network's weights, but also every single node in the network computes a subnetwork function. We will call the subnetwork function computed by the i-th node just F_i. At the tip of an edge w_{ij} going from node i to node j the subnetwork function $F_{ij} = w_i F_i$ is computed. We can look at any subnetwork function F_{ij} in a neural network, select a weight w_k, and compute $\partial F_{ij}/\partial w_k$. We call the union of all paths going backward from the point where F_{ij} is produced to the edge with associated weight w_k the *backpropagation path* between these two points, and the result of doing backpropagation over this path its *backpropagation path value.*

Summarizing: the backpropagation algorithm can be reduced to the following prescription:

- Feed-forward step: transport information through the network and compute at each node its associated one-dimensional function f and its derivative f'. Store the first value on the right side of the node and the second on the left side.
- Backpropagation: from the network's output node, with network function F, being considered go back to the selected weight w_{ik}. To compute $\partial F/\partial w_{ik}$ collect multiplicatively all left sides of the nodes found on the backward path. When a backward path splits, follow each path independently using the partial result accumulated so far. When backward paths meet add the partial results for each path and continue. Multiply finally with F_i (the input to the edge with weight w_{ik})

3 Second order derivatives

Now we set the stakes higher and investigate the case of second order derivatives, that is expressions of the form $\partial^2 F/\partial w_i \partial w_j$, where F is the network function as before and w_i and w_j are network's weights. We can think of each weight as a small potentiometer and we want to find out what happens to the network function when the resistivity of both potentiometers is varied.

Figure 4 shows the general case. Let us assume without loss of generality that the input to the network is the one-dimensional value x. The network function F is computed at the output node with label q (shown shaded) for the given input value. We can think of the inputs to the output node also as network functions computed by subnetworks of the original network. Let us call these functions $F_{\ell_1 q}, F_{\ell_2 q}, \ldots, F_{\ell_m q}$. If the one-dimensional function at the output node is g, the network function is the composition

$$F(x) = g(F_{\ell_1 q}(x) + F_{\ell_2 q}(x) + \ldots + F_{\ell_m q}(x))$$

We are interested in computing $\partial^2 F(x)/\partial w_i \partial w_j$ for two given network weights w_i

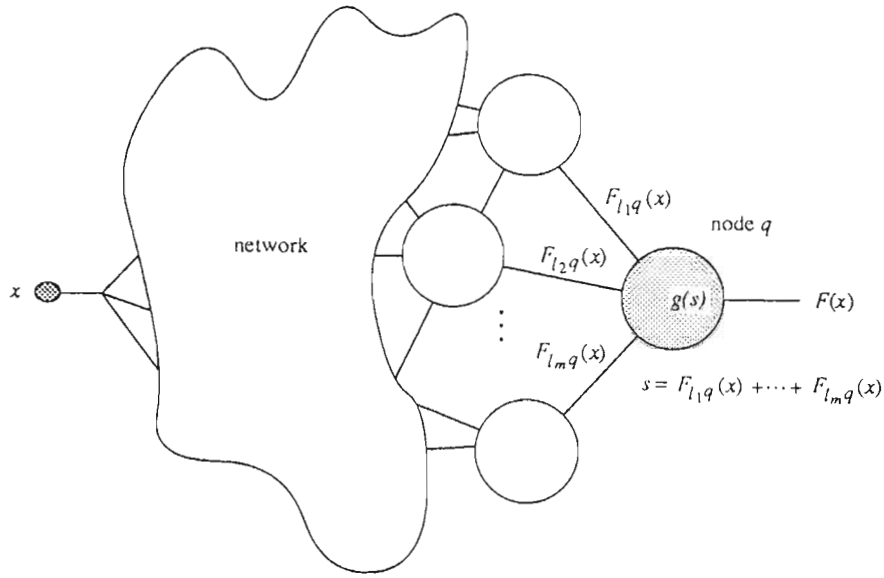

Figure 4: Second order calculation

and w_j. Simple differential calculus tells us that

$$\frac{\partial^2 F(x)}{\partial w_i \partial w_j} = g''(s)\frac{\partial s}{\partial w_i}\frac{\partial s}{\partial w_j} + g'(s)\left(\frac{\partial^2 F_{\ell_1 q}(x)}{\partial w_i \partial w_j} + \cdots + \frac{\partial^2 F_{\ell_m q}(x)}{\partial w_i \partial w_j}\right)$$

where $s = F_{\ell_1 q}(x) + F_{\ell_2 q}(x) + \ldots + F_{\ell_m q}(x)$. This means that the desired second order partial derivative consists of two terms: the first one is the second derivative of g evaluated at its input multiplied with the partial derivatives of the sum of the m subnetwork functions $F_{\ell_1 q}, \ldots, F_{\ell_m q}$, once with respect to w_i and once with respect to w_j. The second term is the first derivative of g multiplied with the sum of the second order partial derivatives of each subnetwork function with respect to w_i and w_j. We call this term the *second order correction*. The recursive structure of the problem is immediately obvious. We already have an algorithm to compute the first partial derivatives of any network function with respect to a weight. We just need to use the above expression in a recursive manner to obtain the second order derivatives we want.

We thus extend the feed-forward labeling phase of the backpropagation algorithm in the following manner: at each node which computes a one-dimensional function f we will store *three* values: $f(x), f'(x)$ and $f''(x)$, where x represents the input to this node. When looking for the second order derivatives we apply the recursive strategy given above.

4 Examples of second-order backpropagation

Consider the network shown in Figure 5, commonly used to compute the XOR function. The left node is labeled 1, the right node is labeled 2. The input values x and y are kept fixed and we are interested in the second order partial derivative

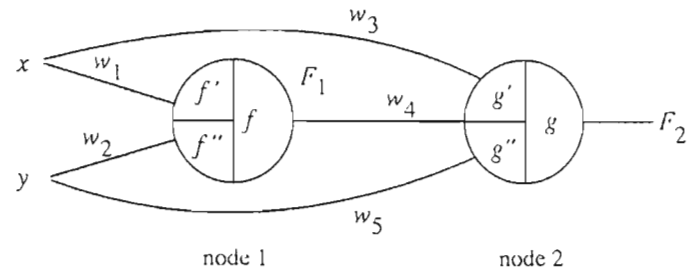

Figure 5: A two neuron network

of the network function $F_2(x, y)$ with respect to the weights w_1 and w_2. By mere inspection and using the recursive method mentioned above, we see that the first term of $\partial^2 F_2/\partial w_1 \partial w_2$ is the expression

$$g''(w_3x + w_5y + w_4f(w_1x + w_2y))(w_4f'(w_1x + w_2y)x)(w_4f'(w_1x + w_2y)y).$$

In this expression $(w_4f'(w_1x + w_2y)x)$ is the backpropagation path value from the output of the node which computes the function f, including the multiplication with the weight w_4 (that is the subnetwork function w_4F_1), up to the weight w_1. The term $(w_4f'(w_1x + w_2y)y)$ is the result of backpropagation for w_4F_1 up to w_2. The second order correction needed for the computation of $\partial^2 F_2/\partial w_1 \partial w_2$ is

$$g'(w_3x + w_5y + w_4f(w_1x + w_2y))\frac{\partial^2 w_4F_1}{\partial w_1 \partial w_2}.$$

Since it is obvious that

$$\frac{\partial^2 w_4F_1}{\partial w_1 \partial w_2} = w_4\frac{\partial^2 F_1}{\partial w_1 \partial w_2} = w_4f''(w_1x + w_2y)xy$$

we finally get

$$\begin{aligned}\frac{\partial^2 F_2}{\partial w_1 \partial w_2} &= g''(w_3x + w_5y + w_4f(w_1x + w_2y))(w_4f'(w_1x + w_2y)x)(w_4f'(w_1x + w_2y)y) \\ &\quad + g'(w_3x + w_5y + w_4f(w_1x + w_2y))w_4f''(w_1x + w_2y)xy.\end{aligned}$$

The reader can *visually* check the following expression:

$$\frac{\partial^2 F_2}{\partial w_1 \partial w_5} = g''(w_3x + w_5y + w_4f(w_1x + w_2y))(w_4f'(w_1x + w_2y)x)y.$$

In this case no second order correction is needed, since the backpropagation paths up to w_1 and w_5 do not intersect. What is the general strategy to make these kind of computations? Figure 6 shows the main idea:

- Perform the feed-forward labeling step in the usual manner, but store additionally at each node the second derivative of the node's function evaluated at its input

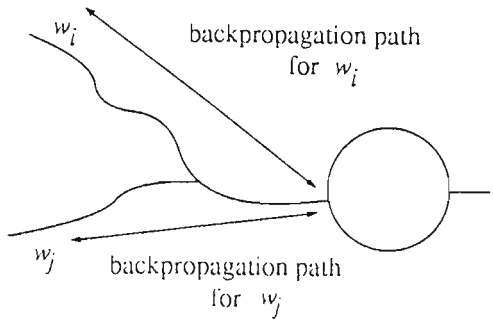

Figure 6: Intersecting paths to a node

- Select two weights w_i and w_j and an output node whose associated network function we want to derive. The second order partial derivative of the network function with respect to these weights is: the product of the stored g'' value with the backpropagation path value from the output node up to weight w_i and with the backpropagation path value from the output node up to weight w_j. If the backpropagation path for w_i and w_j intersect, a second order correction is needed which is equal to the stored value of g' multiplied with the sum of the second order derivative with respect to w_i and w_j of all subnetwork function inputs to the node which belong to intersecting paths.

This looks like an intricate rule, but it is again the chain rule for second order derivatives expressed in a recursive manner. Consider the multilayer perceptron shown in Figure 7. A weight w_{ih} in the first layer of weights and a weight w_{jm} in the second layer can only interact at the output node m. The second derivative of F_m with respect to w_{ih} and w_{jm} is just the stored value f'' multiplied with the stored output of the hidden neuron j and the backpropagation path up to w_{ih}, that is $w_{hm}h'x_i$. Since the backpropagation paths for w_{ih} and w_{jm} do not intersect this is the required expression. This is also the expression found analytically by Bishop [1993]

In the case that one weight lies in the backpropagation path of another a simple adjustment has to be done. Let us assume that weight w_{ik} lies in the backpropagation path of weight w_j. The second order backpropagation algorithm is performed as usual and the backward computation proceeds up to the point where weight w_{ik} transports an input to a node k for which a second order correction is needed. Figure 8 shows the situation. The information transported through the edge with weight w_{ik} is the subnetwork function F_{ik}. The second order correction for the node with primitive function g is

$$g' \frac{\partial^2 F_{ik}}{\partial w_{ik} \partial wj} = g' \frac{\partial^2 w_{ik} F_i}{\partial w_{ik} \partial wj} =$$

but this is simply

$$g' \frac{\partial F_i}{\partial w_j}$$

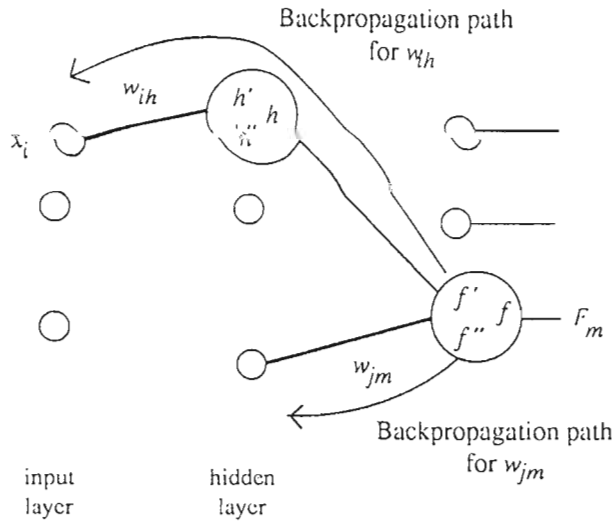

Figure 7: Multilayer perceptron

since the subnetwork function F_i does not depend on w_{ik}. Thus, the second order backpropagation method must be complemented by the following rule:

- If the second order correction to a node k with activation function g involves a weight w_{ik} (that is, a weight directly affecting node k) and a node w_j, the second order correction is just g' multiplied with the backpropagation path value of the subnetwork function F_i with respect to w_j.

5 Explicit calculation of the Hessian

For the benefit of the reader we put together in this section all the pieces of what we call second order backpropagation. We consider the case of a single input pattern into the network, since the more general case is easy to handle.

Second order backpropagation

- Extend the neural network by adding nodes which compute the squared difference of each component of the output and the expected target values. Collect all this differences at a single node, whose output is the error function of the network. The activation function of this node is the identity.
- Label all nodes in the feed-forward phase with the result of computing $f(x)$, $f'(x)$ and $f''(x)$, where x represents the global input to each node and f its associated activation function.
- Starting from the error function node in the extended network, compute the second order derivative of E with respect to two weights w_i and w_j, by proceeding recursively in the following way.
  - The second order derivative of the output of a node G with activation function g with respect to two weights w_i and w_j is the product of the

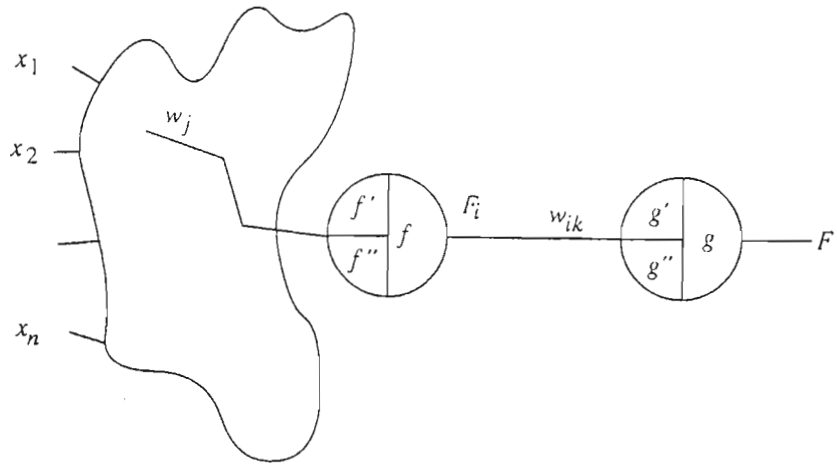

Figure 8: The special case

stored g'' value with the backpropagation path values between w_i and the node G and between w_j and the node G. A second order correction is needed if both backpropagation paths intersect.

- The second order correction is equal to the product of the stored g' value with the sum of the second order derivative (with respect to w_i and w_j) of each node whose output goes directly to G and which belongs to the intersection of the backpropagation paths of w_i and w_j.
- In the special case that one of the weights, for example, w_i connects node h directly to node G, the second order correction is just g' multiplied with the backpropagation path value of the subnetwork function F_h with respect to w_j.

A final example is the calculation of the whole Hessian matrix for the network shown in the previous section (Figure 5). We omit the error function expansion and compute the Hessian of the network function F_2 with respect to the five network's weights. The labelings of the nodes are f, f', and f'' computed over the input $w_1x + w_2y$, and g, g', g'' computed over the input $w_4f(w_1x + w_2y) + w_3x + w_5y$. Under this assumptions the components of the upper triangular part of the Hessian are the following:

$$
\begin{aligned}
H_{11} &= g''w_4^2f'^2x^2 + g'w_4f''x^2 \\
H_{22} &= g''w_4^2f'^2y^2 + g'w_4f''y^2 \\
H_{33} &= g''x^2 \\
H_{44} &= g''f^2 \\
H_{55} &= g''y^2 \\
H_{12} &= g''w_4^2f'^2xy + g'w_4f''xy \\
H_{13} &= g''w_4f'x^2 \\
H_{14} &= g''w_4f'xf + g'f'x
\end{aligned}
$$

$$
\begin{aligned}
H_{15} &= g'' w_4 f' x y \\
H_{23} &= g'' w_4 f' y x \\
H_{24} &= g'' w_4 f' y f + g' f' y \\
H_{25} &= g'' w_4 f' y^2 \\
H_{34} &= g'' x f \\
H_{35} &= g'' x y \\
H_{45} &= g'' y f
\end{aligned}
$$

All these results were obtained by simple inspection of the network shown in Figure 5. Note that the method is totally general, in the sense that each node can compute a different activation function.

6 Conclusions

We have shown in this paper how to compute the Hessian matrix in an efficient way by mere inspection of the neural network being considered. With some experience it is easy to compute the Hessian even for convoluted feed-forward topologies. This can be done either symbolically or numerically. The importance of this result lies in that once the recursive strategy has been defined it is easy to implement it in a computer. It is the same kind of difference as the one existing between the chain rule and the backpropagation algorithm. The first one gives us the same result as the second, but backpropagation tries to organize the data in such a way that redundant computations are avoided. This can be done also for the method described in this paper. The calculation of the Hessian matrix involves the repeated computation of the same terms. The network itself provides us in this case with a data structure in which we can store partial results and with which we can organize the computation. This explains why standard and second order backpropagation are also of interest for computer algebra systems. It is not very difficult to program the method described here in a way that minimizes the number of arithmetic operations needed. The key observation is that the backpropagation path values can be stored to be used repetitively and that the nodes in which the backpropagation paths of different weights intersect need to be calculated only once. It is then possible to optimize the computation of the Hessian using graph traversing algorithms.

A final observation is that computing the diagonal of the Hessian matrix involves only local communication in a neural network. Since the backpropagation path to a weight intersects itself in its whole length, the computation of the second partial derivative of the associated network function of an output neuron with respect to a given weight can be organized as a recursive backward computation over this path. Pseudo-Newton methods [Becker, le Cun 89] used by some learning algorithms can profit from this computational locality.

References

[Battiti 92] Roberto Battiti, "First- and Second-Order Methods for Learning: Between Steepest Descent and Newton's Method", *Neural Computation*, Vol. 4, 1992, pp. 141-166.

[Becker, le Cun 89] Sue Becker and Yann le Cun, "Improving the Convergence of Back-Propagation Learning with Second Order Methods", in: D. Touretzky, G. Hinton and T. Sejnowski (eds.), *Proceedings of the 1988 Connectionist Models Summer School*, Morgan Kaufmann Publishers, 1989.

[Bishop 92] Chris Bishop, "Exact Calculation of the Hessian Matrix for the Multilayer Perceptron", *Neural Computation*, Vol. 4, 1992, pp. 494-501.

[Rojas 93a] Raúl Rojas, *Theorie der neuronalen Netze*, Springer-Verlag, Berlin, 1993.

[Rojas 93b] Raúl Rojas, "A Graphical Proof of the Backpropagation Learning Algorithm", in V. Malyshkin (ed.), *Parallel Computing Technologies, PACT 93*, Obninsk, 1993.