9413

Backpropagation without tears

Raúl Rojas†
Mathematics and Computer Science Department
Free University of Berlin, Takustr. 9
Berlin, 14195, Germany

ABSTRACT

Backpropagation is the most popular training method for multilayer neural networks. Learning in this kind of systems consists of minimizing the error function for a given training set. Since this function is continuous and differentiable, an iterative gradient descent method can be used to find its local minima. We give in this paper a new proof of the backpropagation algorithm using a graphical approach in which the calculation of the gradient of the error function reduces to a graph labelling problem. This method is not only more general than the usual analytical derivations, which handle only the case of special network topologies, but also much easier to follow and understand. It shows also how the algorithm could be efficiently implemented in computing systems in which only local information can be transported through the network.

Keywords: Backpropagation; Learning in Neural Networks; Graph Labelling.
CR Classification Scheme: I.2.6; F.2.0

Introduction

Neural networks represent an alternative computational paradigm, which has received much attention in the last years [1],[5]. They differ in several important ways from conventional computer systems. On the one hand, neural networks are massively parallel systems in which information flows simultaneously through a set of computing units. On the other hand, we do not want to program them in the way we do with conventional computers. Neural networks should learn to perform a computation by analyzing a finite set of examples of input-output pairs. The system should adjust its parameters, so that the network of computing units learns to reproduce the desired output as closely as possible.

† e-mail: rojas@inf.fu-berlin.de

Moreover, the network should be able to *generalize*, in the sense that unknown inputs are to be mapped to new interpolated outputs. In this case our system constitutes a *mapping network* which maps neighborhoods of the known inputs onto neighborhoods of the known outputs.

The most popular approach to implement learning in neural networks is the backpropagation algorithm, a gradient descent method Although the idea behind backpropagation is rather simple, the published derivations of the algorithm are unnecessarily clumsy [5] or they achieve elegance by using high-powered differential operators of the kind that computer science students certainly are not familiar with.

We show in this paper that backpropagation can be very easily derived by thinking of the gradient calculation as a graph labelling problem. This approach is not only elegant, but also more general than the traditional derivations found in most textbooks. General network topologies are handled right from the beginning, so that the proof of the algorithm is not reduced to the multilayered case. Thus you can have it both ways: more general yet simpler, i.e. *backpropagation without tears*.

In this paper we give first a short introduction to the neural network computational paradigm and its associated learning problem, and from there we derive the backpropagation algorithm using our graph labelling approach.

Computing with Neural Networks

We define a neural network as a *computational graph*, whose nodes are computing units and whose directed edges transmit numeric information from node to node. Each computing unit is capable of evaluating a single primitive function of its input. In most neural network models each unit computes the same primitive function, but other arrangements are also possible. Each edge of the graph has an associated weight, which is multiplied by the numeric information being transmitted. The *input units* are those nodes which receive the external numeric information to be processed by the network. The *output units* are those nodes whose results are transmitted outside of the network. It is these results which interest us. In networks without cycles, also called *feed-forward networks*, the evaluation order given by the connection pattern of the network is unambiguous, so that we do not need to synchronize the computing units. If more than two edges coincide on a node, the numeric information they convey is added before the node evaluates its associated one-dimensional primitive function. Fig. 1 shows an example of a feed-forward neural network. The network represents in fact a chain of function compositions which transforms an input to an output vector (called a *pattern*). The network is just an implementation of a composite function from the input to the output space, which we call the *network function*.

For a given network the network function depends on the primitive functions implemented at the nodes, the topology of the network and the weights of the edges. For a fixed topology and a fixed class of computing units, the network function varies with each

choice of network weights. We say that the network function is *parameterized* by the network weights. The learning problem consists in finding the optimal combination of weights such that the network function F approximates a given function f as closely as possible.

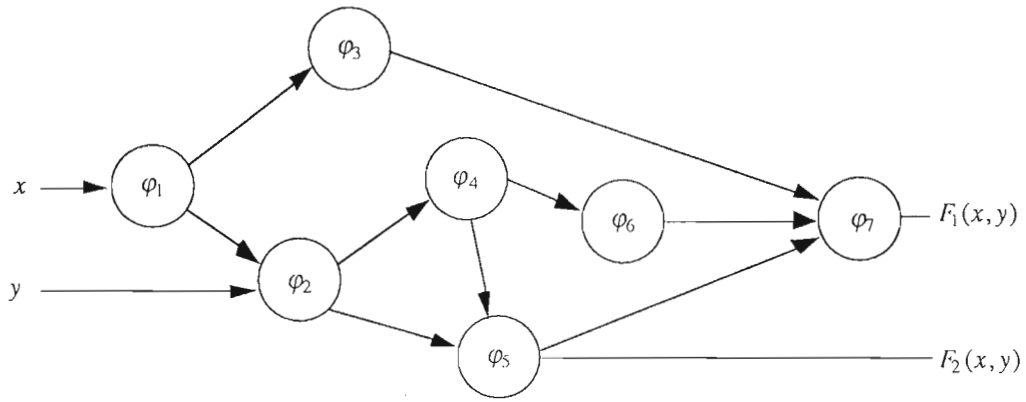

Fig. 1. Example of a feed-forward network

Neural networks are in some sense a generalization of some traditional methods of function approximation. If we are given a continuous real function $f:[0,1]\rightarrow R$, we know from the Weierstrass theorem that we can approximate f uniformly with a polynomial of degree n, and that the higher the degree of the polynomial, the better the approximation we can get. Fig. 2 shows a network capable of computing the Weierstrass approximation, when the coefficients $a_0, a_1, ..., a_n$ of the approximating polynomial are known. Stated in this way the problem seems trivial. Usually though, we are not given the function f explicitly but only implicitly through some examples. We are provided with a set of m input-output pairs $(x_1, f(x_1))$, $(x_2, f(x_2)), ..., (x_m, f(x_m))$ and then we try to find the weights which minimize the squared error of the approximation produced by the network. In this case the best solution is the one given by the well known least-squares method developed by Gauss.

By using more than one layer of computing units, the number of nodes needed for an approximation can be dramatically reduced. We are also interested in using other than polynomials as activation functions of the units; we want in fact to develop a method capable of finding the weights needed in a network of arbitrary differentiable activation functions. In this case, it is very difficult to analytically minimize the squared error for the training set. An iterative gradient descent method has to be used and this brings us to the core of the problem: When provided with a network of primitive functions, how do we find the gradient of the network function according to the weights of the network? The answer to this problem is the backpropagation algorithm.

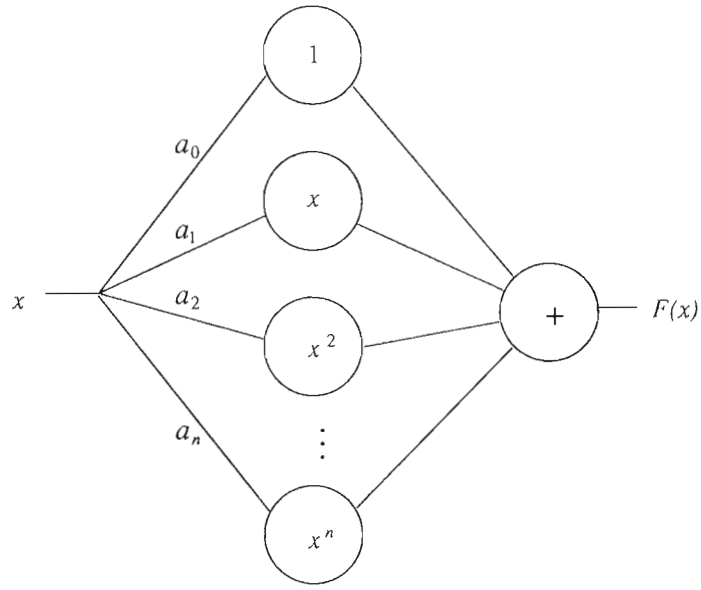

Fig. 2. A polynomial network

Learning in Neural Networks

Consider a feed-forward network with n input and m output units. It can consist of any number of hidden units and can exhibit any desired feed-forward connection pattern. We are also given a training set $(\mathbf{x}_1,\mathbf{t}_1),(\mathbf{x}_2,\mathbf{t}_2),...,(\mathbf{x}_p,\mathbf{t}_p)$ consisting of p ordered pairs of n- and m-dimensional vectors, which are called the input and output patterns respectively. Let the primitive functions calculated at each node of the network be continuous and differentiable. The weights of the edges are real numbers selected at random. When the input pattern $\mathbf{x}_i$ from the training set is presented to this network, it produces an output $\mathbf{o}_i$ different in general from $\mathbf{t}_i$. What we want, is to make $\mathbf{o}_i$ and $\mathbf{t}_i$ identical for $i=1,...,p$ by using a learning algorithm. More precisely, we want to minimize the *error function* of the network, defined as

$$E=(1/2)\sum_{i=1}^{p}\left|\mathbf{o}_i-\mathbf{t}_i\right|^2.$$

The first step of the minimization process consists in extending the network, so that it computes the error function automatically. Fig. 3 shows how this is done. Every output unit $j=1,...,m$ of the network is connected to a node which evaluates the function $(1/2)(o_{ij}-t_{ij})^2$, where o_{ij} and t_{ij} denote the j-th component of the output vector $\mathbf{o}_i$, respectively the target $\mathbf{t}_i$. The output of the new m nodes is collected at a node which just adds them up and gives the sum as its output. The same extension has to be done for each pattern $\mathbf{t}_i$. A computing unit collects all quadratic errors and outputs their sum. The output of this extended network is the error function E.

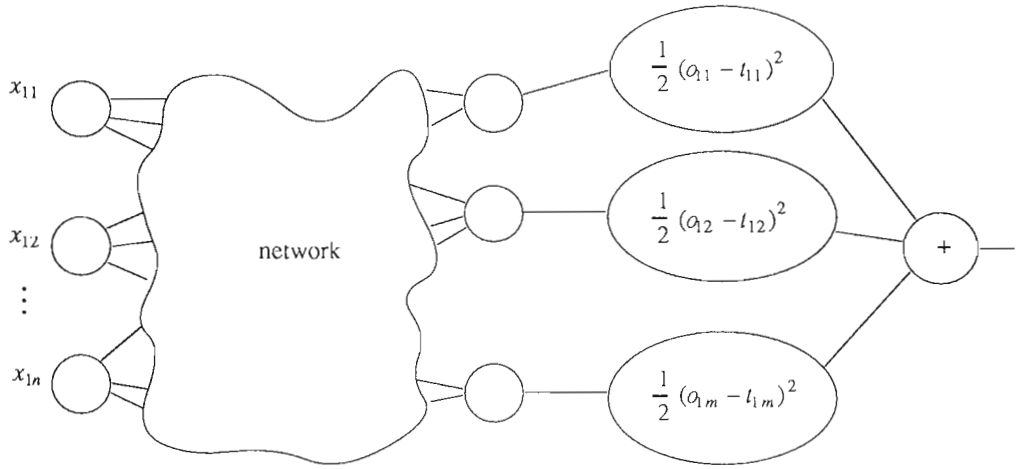

Fig. 3. Extended network for the computation of the error function

We have now a network capable of calculating the error function for a given training set. The weights in the network are the only parameters that can be changed. We can tune them, trying to make the quadratic error E as low as possible. Since E is calculated by the extended network exclusively through composition of the node functions, it is a continuous and differentiable function of the ℓ weights $w_1, w_2, \ldots, w_\ell$ of the network. We can thus minimize E by using an iterative process of gradient descent, for which we need to calculate the gradient

$$\nabla E = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_\ell} \right)$$

and adjust each weight w_i by using the increment

$$\Delta w_i = -\gamma \frac{\partial E}{\partial w_i}, \quad i = 1, \ldots, \ell,$$

where γ represents a learning constant, i.e. a proportionality parameter which defines the step length of each iteration in the negative gradient direction.

With this extension of the original network the whole learning problem reduces now to the question of calculating the gradient of a network function with respect to its weights. Once we have a method to compute this gradient, we can adjust the network weights iteratively. In this way we expect to find a minimum for the error function, where $\nabla E = 0$.

Derivatives of Network Functions

Forget now everything about training sets and learning. Our objective is just to find a method for calculating efficiently the gradient of a one-dimensional network function according to the weights of the network. Because the network is equivalent to a complex chain of function compositions, we expect the chain rule of differential calculus to play a major role in finding the gradient of the function. We take account of this fact by giving the nodes of the network a composite structure. Each node consists now of a left and a right side. The right side computes the primitive function associated with the node, whereas the left side computes the derivative of this primitive function for the same input. The network is evaluated now in two stages: in the first one, the feed-forward step, information comes from the right and each unit evaluates its primitive function f in its right side as well as the derivative of f in its left side. Both results are stored in the unit, but only the result from the right side is given off and transmitted to the connected units. The second step, the backpropagation step, consists in running the whole network *backwards*, whereby the results stored in the left side are now used. There are three main cases which we have to consider.

•*First case: function composition*

The network of Fig. 4 contains only two nodes. In the feed-forward step, incoming information into a unit is used as the argument for the evaluation of the node's primitive function and its derivative. The network computes in this step the composition of the functions f and g. Fig. 5 shows the state of the network *after* the feed-forward step. The correct result of the function composition has been produced at the output neuron and each neuron has stored some information in its left side.

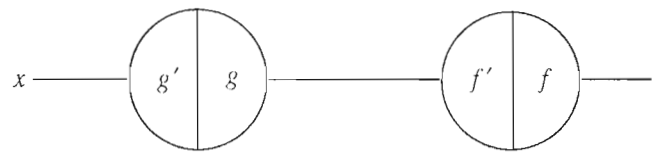

Fig. 4. Network for the composition of two functions

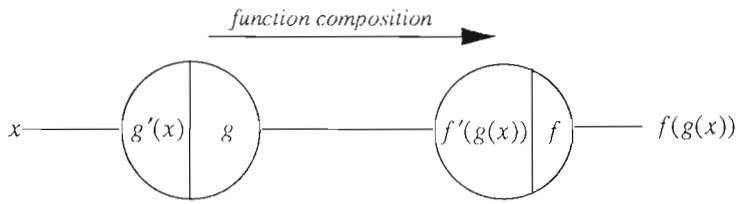

Fig. 5. Result of the feed-forward step

In the backpropagation step the input from the right of the network is the real constant 1. Incoming information to a node is *multiplied* by the value stored in its left side. The result of the multiplication is given off to the left and the information is transported to the next unit. We call the result at each node the traversing value at this node. Fig. 6 shows the final result of the backpropagation step, which is $f'(g(x))g'(x)$, that is the derivative of the function composition $f \circ g$ implemented by this network. The backpropagation step provides us with an implementation of the chain-rule. Any sequence of function compositions can be evaluated in this way and its derivative can be obtained in the backpropagation step.

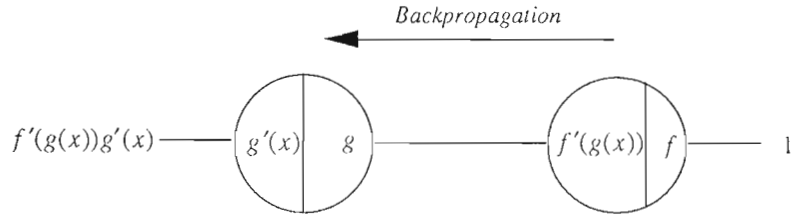

Fig. 6. Result of the backpropagation step

•*Second case: function addition*

The next case to consider is the addition of two primitive functions. Fig. 7 shows a network to compute the addition of the functions f_1 and f_2. The additional node has been included just to handle the addition of the two functions. Its activation function is the identity, whose derivative is 1. In the feed-forward step the network computes the result $f_1(x)+f_2(x)$. In the backpropagation step the constant 1 is fed from the left side into the network. All incoming edges to a network fan-out the traversing value at this node and distribute it to the connected neurons. Where two paths meet the computed traversing values are added. Fig. 8 shows the result of the backpropagation step for the network. The result is the derivative of the function addition $f_1 + f_2$. A simple proof by induction shows that the derivative of the addition of any number of functions can be handled in the same way.

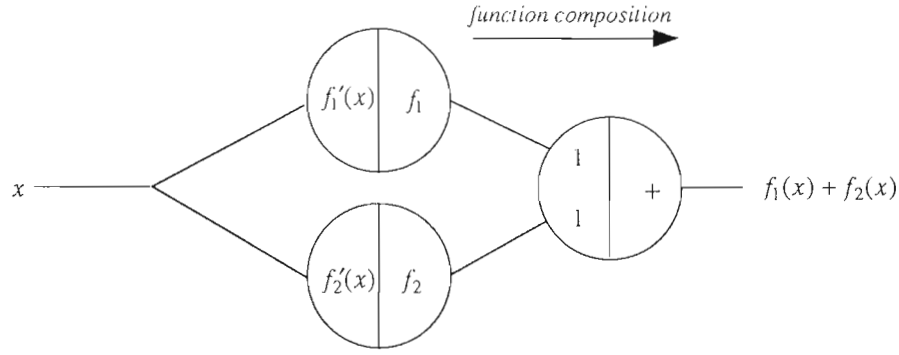

Fig. 7. Addition of functions

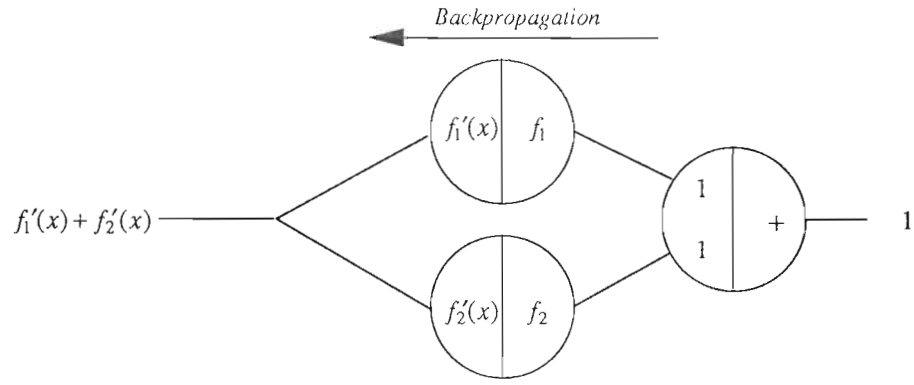

Fig. 8. Result of the backpropagation step

•*Third case: weighted edges*

The last case we have to consider is weighted edges. In the feed-forward step the incoming information x is multiplied by the edge's weight w. The result is wx. In the backpropagation step the traversing value 1 is multiplied by the weight of the edge. The result is w, which is the derivative of wx with respect to x. We conclude from this case that weighted edges are used in exactly the same way in both steps: they modulate the information transmitted in each direction by multiplying it by the edges' weights.

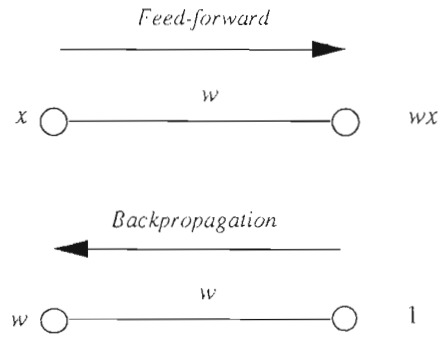

Fig. 9. Backpropagation at an edge

Steps of the Backpropagation Algorithm

We can now formulate the complete backpropagation algorithm and give a proof by induction that it works in arbitrary feed-forward networks with differentiable activation functions at the nodes. We assume that we are dealing with a network with a single input and a single output unit. The two phases of the algorithm are the following:

Feed-forward step: the input x is fed into the network. The primitive functions at the nodes and their derivatives are evaluated at each node. The derivatives are stored.

Backpropagation step: the constant 1 is fed into the output unit and the network is run backwards. Incoming information to a node is added and the result is multiplied by the value stored in the left part of the unit. The result is given off to the left of the unit. The result collected at the input unit is the derivative of the network function with respect to x.

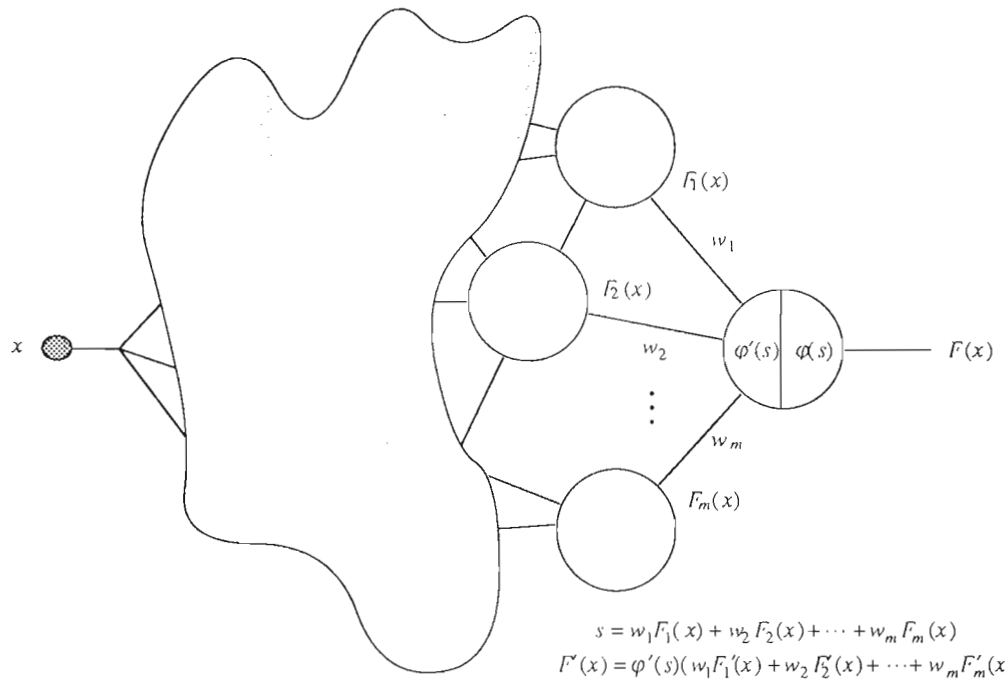

Fig. 10. Backpropagation at the last node

We showed before that the algorithm works for units in series, units in parallel and also when weighted edges are present. Let us make the induction assumption that the algorithm works for any feed-forward network with n or less that n nodes. Consider now the network of Fig. 10, which is made of $n+1$ nodes. The feed-forward step is first executed and the result of the output unit is the network function F evaluated at x. Assume that m units, whose respective outputs are $F_1(x)$, $F_2(x),...,F_m(x)$, are connected to the output unit. Since the primitive function of the output unit is φ, we know that

$$F(x) = \varphi(w_1F_1(x) + w_2F_2(x) + \cdots + w_mF_m(x)).$$

The derivative of F at x is thus

$$F'(x) = \varphi'(s)(w_1F_1'(x) + w_2F_2'(x) + \cdots + w_mF_m'(x)).$$

where $s = w_1 F_1(x) + w_2 F_2(x) + \cdots + w_m F_m(x)$. The subgraph of the main graph which includes all possible paths from the input unit to the unit whose output is $F_1(x)$, defines a subnetwork whose network function is F_1 and which consists of n or less units. By the induction assumption we can calculate the derivative of F_1 at x, by introducing a 1 into the unit and running the subnetwork backwards. The same can be done with the units whose output is $F_2(x),...,F_m(x)$. If we introduce the constant $\varphi(s)w_1$ instead of a 1, we get at the input unit in the backpropagation step $w_1 F_1'(x)\varphi(s)$ and $w_2 F_2'(x)\varphi(s),...,w_m F_m'(x)\varphi(s)$ with the rest of the units. But in the backpropagation step with the whole network we add these m results and we finally get

$$\varphi'(s)(w_1 F_1'(x) + w_2 F_2'(x) + \cdots + w_m F_m'(x))$$

which is the derivative of F evaluated at x. Note that introducing the constants $w_1 F_1'(x)\varphi(s),...,w_m F_m'(x)\varphi(s)$ into the m units connected to the output unit can be done by introducing a 1 into the output unit, multiplying by the stored value $\varphi(s)$ and distributing the result to the m units through the edges with weights $w_1, w_2,..., w_m$. We are in fact running the network backwards as the backpropagation algorithm demands. This means that the algorithm works with networks of $n+1$ nodes and this concludes our proof.

The backpropagation algorithm still works correctly for networks with more than one input unit in which several independent variables are involved. In a network with two inputs for example, where the independent variables x and y are fed into the network, the network result can be called $F(x, y)$. The network function now has two arguments and we can compute the partial derivative of F with respect to x or with respect to y. The feed-forward step remains unchanged and all left side slots of the units are filled as usual. But in the backpropagation step we can identify two subnetworks: one consists of all paths connecting the first input unit to the output unit and another of all paths from the second input unit to the output unit. By applying the backpropagation step in the first subnetwork we get the partial derivative of F with respect to x at the first input unit. The backpropagation step on the second subnetwork yields the partial derivative of F with respect to y at the second input unit. But note that we can overlap both computations and perform a single backpropagation step over the whole network. We still get the same results.

Learning with Backpropagation

We consider again the learning problem for neural networks. Since we want to minimize the error function E, and this depends on the network weights, we have to deal with each weight in the network one at a time. The feed-forward step is computed in the usual way, but now we also store the output of each neuron in its right side. We perform

the backpropagation step in the extended network used to compute the error function and we then fix our attention on one of the weights, say w_{ij} which points from the i-th to the j-th node in the network. This weight can be treated as an input channel into the subnetwork made of all paths starting at w_{ij} and ending in the single output unit of the network. The information fed into the subnetwork in the feed-forward step was $o_i w_{ij}$, where o_i is the stored output of unit i. The backpropagation step computes the gradient of E with respect to this input, that is $\partial E / \partial o_i w_{ij}$. Since in the backpropagation step o_i is treated as a constant, we finally have

$$\frac{\partial E}{\partial w_{ij}} = o_i \frac{\partial E}{\partial o_i w_{ij}}.$$

The backpropagation step is performed in the usual way. All subnetworks defined by each weight of the network can be handled simultaneously, but we now store a third quantity at each node: the result of the backward computation in the backpropagation step up to this node. We call this quantity the *backpropagated error*. If we denote the backpropagated error at the i-th node by δ_i, we can then write the partial derivative of E with respect to w_{ij} as:

$$\frac{\partial E}{\partial w_{ij}} = o_i \delta_j.$$

Once all partial derivatives have been computed, we can perform gradient descent by adding to each weight w_{ij} the increment

$$\Delta w_{ij} = -\gamma o_i \delta_j.$$

This correction step is what is needed to transform the backpropagation algorithm into a learning method for neural networks.

Final Remarks

We have given a new graphical proof of the backpropagation algorithm essentially simpler than the traditional ones [1],[5], yet more general because it applies to arbitrary feed-forward networks. By understanding backpropagation as a graph labelling algorithm it is much easier to handle complex network topologies. The graphical approach immediately suggests a hardware implementation technique for backpropagation. Our method can be also used to show, among other results, that summation of the inputs to a node is the only integration function for neural networks which guarantees locality of the learning algorithm [3]. By visualizing the learning rule, it is also possible to get a deeper understanding of the hardware and locality requirements of fast variations of

backpropagation [2]. A further step is to compute second-order derivatives using the same kind of approach. This has been shown elsewhere [4].

Acknowledgements

This work was done while visiting the *International Computer Science Institute* in Berkeley. Thanks are due to Marcus Pfister and Joachim Beer for his valuable comments and to Prof. Elfriede Fehr for her support and encouragement.

References

1. Hertz, J., Krogh, A., and Palmer, R. *Introduction to the Theory of Neural Computation*. Addison-Wesley, Redwood City,1991.
2. Pfister, M., and Rojas, R. Speeding-up Backpropagation – A comparison of orthogonal techniques. *International Joint Conference on Neural Networks*, Nagoya, 1993.
3. Rojas, R. *Theorie der neuronalen Netze*. Springer-Verlag, Berlin, 1993.
4. Rojas, R. Second-Order Backpropagation. Technical Report, *International Computer Science Institute*, 1993.
5. Rumelhart D., and J. McClelland, J., Eds. *Parallel Distributed Processing*. MIT Press, Cambridge, Mass., 1986.