For this bound to be smaller than some small number δ we must have

$$N < \frac{\varepsilon^2 p - 8\log(4/\delta)}{8\log(2p)}. \tag{6.80}$$

In the limit of large N and p this gives approximately

$$p > 8N\log(N)/\varepsilon^2. \tag{6.81}$$

Thus the number of training examples needed to train an N-input perceptron and obtain good generalization in the worst case scales up as $N\log N$. This doesn't seem too bad—considering the fact that there are 2^N possible examples—but the prefactor $8/\varepsilon^2$ dampens any enthusiasm for small ε.

As a second example consider a general feed-forward network with M threshold nodes and W weights (including thresholds). Baum and Haussler [1989] calculated an upper bound for the VC dimension of any such network, obtaining

$$d_{VC} \leq 2W\log_2(eM) \tag{6.82}$$

where e is the base of the natural logarithm, $e = \exp(1)$. From this they derived an upper bound on the size p of the training set needed to get a good generalization probability. If the error on the training set is less than $\varepsilon/2$ they showed that at most of the order of $\frac{W}{\varepsilon}\log\frac{M}{\varepsilon}$ examples are needed to obtain a generalization error less than ε.

For a network with N inputs and one fully connected hidden layer of H units they also derived a lower bound

$$d_{VC} \geq 2\lfloor H/2\rfloor N \simeq W \quad \text{(for large } H\text{)} \tag{6.83}$$

where $\lfloor x\rfloor$ means the largest integer not greater than x. They used this to show that one needs (loosely speaking) of the order of W/ε training examples to expect a generalization error less than ε.

From Hertz et al: Intro to the theory of neural computation

6.6 Optimal Network Architectures

We have seen that the network architecture is very important, and each application requires its own architecture. To obtain good generalization ability one has to build into the network as much knowledge about the problem as possible (e.g., the topology of the input space) and limit the number of connections appropriately. It is therefore desirable to find algorithms that not only optimize the weights for a given architecture, but also optimize the architecture itself. This means in particular optimizing the number of layers and the number of units per layer.

Of course there are various different criteria for *optimal*, including generalization ability, learning time, number of units, and so on. In fact, given various hardware restrictions, there may be quite a complicated cost function for the architecture itself. We focus mainly on using as few units as possible; this should not only reduce computational costs and perhaps training time, but should also improve generalization.

It is of course possible to mount a search in the space of possible architectures. We have to train each architecture separately by (say) back-propagation, and then evaluate it with an appropriate cost function that incorporates both performance and number of units. Such a search can be carried out by a **genetic algorithm**, so that good building blocks found in one trial architecture are likely to survive and be combined with good building blocks from other trials [Harp et al., 1990; Miller et al., 1989]. However, this kind of search seems unlikely to be practical for applications requiring large networks, where training just one architecture often requires massive CPU power.

More promising are approaches in which we construct or modify an architecture to suit a particular task, proceeding incrementally. There are two such ways to reach as few units as possible: start with too many and take some away; or start with too few and add some more. We consider examples of each approach.

Pruning and Weight Decay

We have already briefly described one way of optimizing the architecture in the ZIP-code reading network on page 139. There the network was trimmed by removing unimportant connections. It is also possible to prune unimportant *units* [Sietsma and Dow, 1988]. In either case it is necessary to retrain the network after the "brain damage", though this retraining is usually rather fast.

Another approach is to have the network itself remove non-useful connections during training. This can be achieved by giving each connection w_{ij} a tendency to decay to zero, so that connections disappear unless reinforced [Hinton, 1986; Scalettar and Zee, 1988; Kramer and Sangiovanni-Vincentelli, 1989]. The simplest method is to use

$$w_{ij}^{\text{new}} = (1-\varepsilon)w_{ij}^{\text{old}} \tag{6.84}$$

after each update of w_{ij}, for some small parameter ε. This is equivalent to adding a penalty term w_{ij}^2 to the original cost function E_0

$$E = E_0 + \tfrac{1}{2}\gamma\sum_{(ij)} w_{ij}^2 \tag{6.85}$$

and performing gradient descent $\Delta w_{ij} = -\eta\partial E/\partial w_{ij}$ on the resulting total E. The ε parameter is then just $\gamma\eta$.

While (6.85) clearly penalizes use of more w_{ij}'s than necessary, it overly discourages use of large weights; one large weight costs much more than many small ones. This can be cured by using a different penalty term, such as

$$E = E_0 + \tfrac{1}{2}\gamma\sum_{(ij)} \frac{w_{ij}^2}{1+w_{ij}^2} \tag{6.86}$$

which is equivalent to making ε in (6.84) dependent on w_{ij}

$$\varepsilon_{ij} = \frac{\gamma\eta}{\left(1 + w_{ij}^2\right)^2} \tag{6.87}$$

so that small w_{ij}'s decay more rapidly than large ones.

These decay rules perform well in removing unnecessary *connections*, but often we want to remove whole *units*. Then we can start with an excess of hidden units and later discard those not needed. It is easy to encourage this by making the weight decay rates larger for units that have small outputs, or that already have small incoming weights [Hanson and Pratt, 1989; Chauvin, 1989]. For example we could replace (6.87) by

$$\varepsilon_i = \frac{\gamma\eta}{\left(1 + \sum_j w_{ij}^2\right)^2} \tag{6.88}$$

and use this same ε_i for all connections feeding unit i.

Network Construction Algorithms

Rather than starting with too large a network and performing some pruning, it is more appealing to start with a small network and gradually grow one of the appropriate size. There have been several attempts in this direction; we outline three of them and discuss one (the tiling algorithm) in more detail.

Figure 6.16 shows the way that three different algorithms construct networks. In each case the aim is to construct a network that correctly evaluates a Boolean function from N binary inputs to a single binary output, given by a training set of p input-output pairs. We assume that the training set has no internal conflicts (different outputs for the same input). Extensions to multiple outputs are possible, but not discussed here. Threshold units are used in all layers. Each algorithm tries to construct a network using as few units as possible within a particular construction scheme.

Marchand et al. [1990] propose an algorithm than constructs a solution using a single hidden layer, as shown in Fig. 6.16(a). Hidden units are added one-by-one, each separating out one or more of the p patterns, which are then removed from consideration for the following hidden units. Specifically, each hidden unit is chosen so that it has the same output (say +1) for *all* remaining patterns with one target (say +1), and the opposite output (−1) for *at least one* of the remaining patterns with the opposite target (−1); this can always be done. The latter one or more patterns are then removed from consideration for the following hidden units. This process terminates when all remaining patterns have the same target.

The resulting patterns, or **internal representations**, on the hidden layer each have a unique target ±1. Moreover, the hetero-association problem from these internal representations to their targets can be shown to be linearly separable. It can therefore be performed with just one more layer, using the perceptron learning rule.

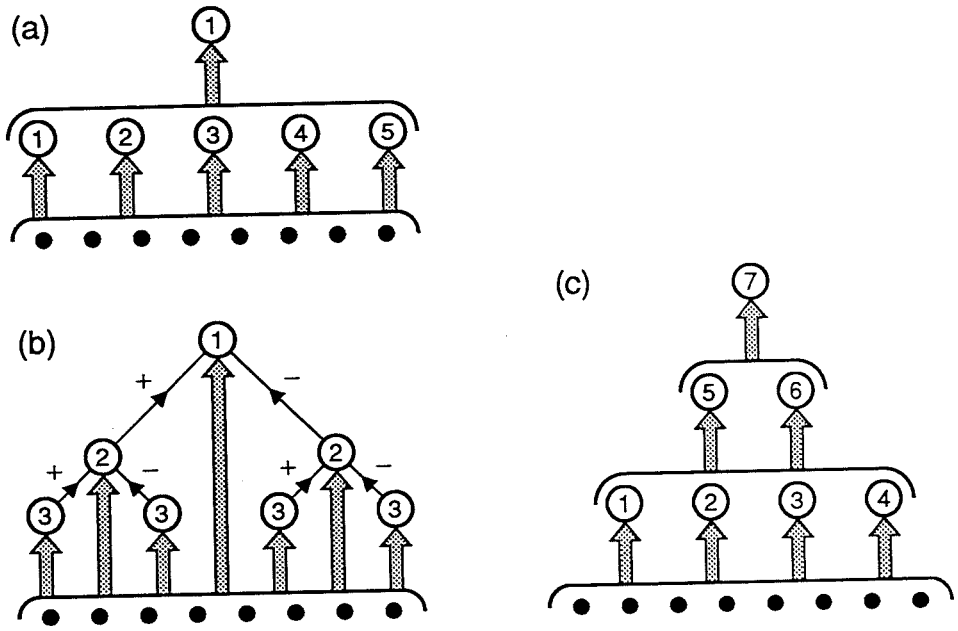


FIGURE 6.16 Network construction algorithms. The black dots are inputs, while the numbered circles are threshold units, *numbered in order of their creation.* Shaded arrows represent connections from all the units or inputs at the arrow's tail. (a) Marchand et al. [1990]. (b) Frean [1990]. (c) Mézard and Nadal [1989].

Figure 6.16(b) shows the kind of architecture generated by the **upstart algorithm** of Frean [1990]. It is specifically for off/on 0/1 units. First we do the best we can with a single output unit 1, directly connected to the input. Then we note all the cases in which the output is wrong, and create two more units 2 (if needed), one to correct the *wrongly on* cases and one to correct the *wrongly off* ones. The subsidiary units 2 are connected to the output units with large positive or negative weights, so that they override the previous output when activated. The subsidiary units 2 are directly connected to the input, and are trained to do the best they can on their own problem of correcting the wrongly on or wrongly off patterns, without upsetting the correct patterns. If necessary we create further units 3 to correct *their* mistakes, and so on. Each additional unit created reduces the number of incorrectly classified patterns by at least one, so the process must eventually cease.

The upstart algorithm generates an unusual hierarchical architecture, but in fact this can be converted into an equivalent two-layer network. All the units of the hierarchical arrangement are placed in the hidden layer, removing the connections between them. Then a new output unit is created and fully connected to the hidden layer; appropriate connections can be found to regenerate the desired targets.

Sirat and Nadal [1990] independently proposed exactly the same way of dividing up the input space. In their approach the "daughter" units are not actually connected to their "parent" unit to correct it, but are used as nodes in a binary decision tree. Note that if the output of a unit in the upstart network is 1 only the daughter unit that corrects *wrongly on* can change this output, and similarly if the output is 0 only the *wrongly-off* daughter can affect it. Therefore, if the units are updated in opposite order—starting from the output unit and working backwards—it is only necessary to update *one* unit at each level. Updating stops when there is no daughter of the right type to correct a unit, and then the last unit that *was* updated determines the class of the input. This procedure corresponds to traversing a binary tree, called a **neural tree** by Sirat and Nadal. Binary decision trees are frequently used for classification, and this one is characterized by having a simple one-layer perceptron at each branching point.

Another interesting algorithm by Fahlman and Lebiere [1990] also builds a hierarchy of hidden units similar to the one in the upstart algorithm. It is called the **cascade-correlation algorithm**, and seems to be very efficient.

Mézard and Nadal [1989] proposed a **tiling algorithm** that creates multilayer architectures such as that shown in Fig. 6.16(c), starting from the bottom and working upwards. Each successive layer has fewer units than the previous one, so the process eventually terminates with a single output unit.

For any such architecture to be successful it is clear that the patterns (internal representations) on every layer must be **faithful representations** of the input patterns. That is, if two input patterns have different targets at the output layer, then their internal representations must be different on every hidden layer. The idea of the tiling algorithm is to start each layer with a **master unit** that does as well as possible on the target task, and then add further **ancillary units** until the representation on that layer is faithful. The next layer is constructed in just the same way, using the output of the previous layer as its input. Eventually a master unit itself classifies all patterns correctly, and is therefore the desired output unit.

The master unit in each layer is trained so as to produce the correct target output (± 1) on as many of its input patterns as possible. This can be done by a variant of the usual perceptron learning rule (5.19) called the **pocket algorithm** [Gallant, 1986]. The unmodified perceptron learning rule wanders through weight space when the problem is *not* linearly separable, spending most time in regions giving the fewest errors, but not staying there. So the pocket algorithm modification consists simply in storing (or "putting in your pocket") the set of weights which has had the longest unmodified run of successes so far. The algorithm is stopped after some chosen time t, which is the only free parameter in the tiling algorithm.

The ancillary units in each layer are also trained using the pocket algorithm, but only on subsets of the patterns. Whenever the representation on the latest layer is *not* faithful, then that layer has at least one activation pattern without a unique target; the subset of input patterns that produces the ambiguous pattern includes both targets. So we train a new ancillary unit on this subset, trying to separate it as far as possible. Then we look again for ambiguous patterns, and repeat as often

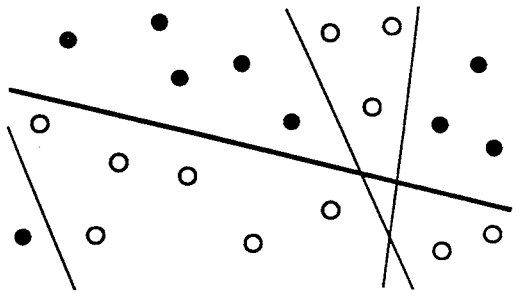


FIGURE 6.17 Tiling the input space. The master unit (heavy line) does the best possible separation of the points. The ancillary units (thinner lines) make sure that the wrongly classified points are separated into classes with the same target.

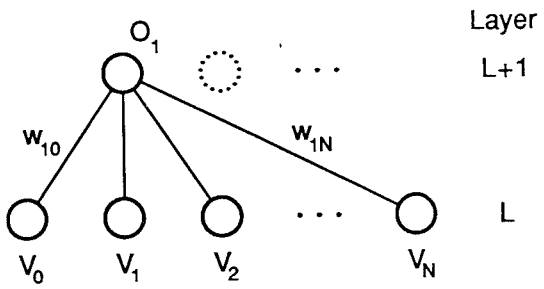


FIGURE 6.18 Notation for convergence proof of the tiling algorithm. V_1 and O_1 are master units, and V_0 is a bias unit.

as necessary until the representation is faithful. Fig. 6.17 shows how this divides up the input space.

To prove that the whole process converges we construct a set of weights that lets the master unit in layer $L+1$ classify at least one more pattern correctly than the master unit in layer L. Of course the pocket algorithm will probably not choose our specific weights, but it will certainly do no worse in the number of misclassifications. Thus each master unit correctly classifies more patterns than the last, and so the network construction process eventually terminates.

We use the notation shown in Fig. 6.18; units (or inputs) V_j in layer L are connected to the master unit O_1 in layer $L+1$ by weights w_{1j}. It is essential to include an explicit bias or threshold, which we do by fixing $V_0 = 1$. We assume that the layer L master unit V_1 classifies q input patterns correctly ($V_1^\mu = \zeta^\mu$), with $q < p$. Now consider a pattern ν that is *not* classified correctly by V_1, so that $V_1^\nu = -\zeta^\nu$. Choose the weights

$$w_{1j} = \begin{cases} 1 & \text{if } j = 1\,; \\ \varepsilon \zeta^\nu V_j^\nu & \text{otherwise} \end{cases} \tag{6.89}$$

with

$$\frac{1}{N} < \varepsilon < \frac{1}{N-2} \tag{6.90}$$

where N is the number of units in layer L besides the bias unit V_0. This makes O_1

classify pattern ν correctly

$$O_1^\nu = \mathrm{sgn}\Big(\sum_j w_{1j} V_j^\nu\Big) = \mathrm{sgn}\big(-\zeta^\nu + \varepsilon\zeta^\nu N\big) = \zeta^\nu \tag{6.91}$$

since $N\varepsilon > 1$. But it also leaves intact the q correct classifications by V_1, for which we find

$$O_1^\mu = \mathrm{sgn}\Big(\sum_j w_{1j} V_j^\mu\Big) = \mathrm{sgn}\Big(\zeta^\mu + \varepsilon\zeta^\nu \sum_{j\neq 1} V_j^\mu V_j^\nu\Big). \tag{6.92}$$

Because $(N-2)\varepsilon < 1$, the second term in the sign function could only change the correct sign of the first term ζ^μ if $|X| = N$ where

$$X = \sum_{j\neq 1} V_j^\mu V_j^\nu . \tag{6.93}$$

But X cannot be $-N$ because $V_0^\mu = V_0^\nu = 1$, and it cannot be $+N$ because V_j^μ and V_j^ν are not identical if the level L representation is faithful. So the q originally correct patterns are not upset, and the new unit O_i classifies at least $q+1$ patterns correctly, as claimed.

It is not yet clear which of the three methods described is best for a given problem. All of them have given encouraging results on simple test problems, both in terms of generalization and in terms of finding efficient architectures. In one comparative test the upstart algorithm used fewer units than the tiling algorithm [Frean, 1990], but more wide-ranging studies are needed. It is also likely that further construction algorithms will be proposed in the future.

The preceding chapter was concerned strictly with supervised learning in *feed-forward* networks. We now turn to supervised learning in more general networks, with connections allowed both ways between a pair of units, and even from a unit to itself. These are usually called **recurrent networks**. They do not necessarily settle down to a stable state even with constant input. Symmetric connections ($w_{ij} = w_{ji}$) ensure a stable state of course (as seen in the Hopfield networks), and the Boltzmann machines discussed first are limited to the symmetric case. In the second section we consider networks without the symmetry constraint, but only treat those that *do* reach a stable state. Then we examine networks that can learn to recognize or reproduce time sequences; some of these produce cyclic output rather than a steady state. We conclude with a discussion of reinforcement learning in recurrent and non-recurrent networks.

7.1 Boltzmann Machines

Hinton and Sejnowski [Hinton and Sejnowski, 1983, 1986; Ackley, Hinton and Sejnowski, 1985] introduced a general learning rule applicable to any *stochastic* network with *symmetric* connections, $w_{ij} = w_{ji}$. They called this type of network a **Boltzmann machine** because the probability of the states of the system is given by the Boltzmann distribution of statistical mechanics. Boltzmann machines may be seen as an extension of Hopfield networks to include hidden units. Just as in feed-forward networks with hidden units, the problem is to find the right connections to the hidden units without knowing from the training patterns what the hidden units should represent.

From Fahlman: 'An Empirical Study of Learning Speed in Back-Propagation Networks'. (Full paper available @ Neuroprose)

3.4. The Quickprop Algorithm

Back-propagation and its relatives work by calculating the partial first derivative of the overall error with respect to each weight. Given this information we can do gradient descent in weight space. If we take infinitesimal steps down the gradient, we are guaranteed to reach a local minimum, and it has been empirically determined that for many problems this local minimum will be a global minimum, or at least a "good enough" solution for most purposes.

Of course, if we want to find a solution in the shortest possible time, we do not want to take infinitesimal steps; we want to take the largest steps possible without overshooting the solution. Unfortunately, a set of partial first derivatives collected at a single point tells us very little about how large a step we may safely take in weight space. If we knew something about the higher-order derivatives -- the curvature of the error function -- we could presumably do much better.

Two kinds of approaches to this problem have been tried. The first approach tries to dynamically adjust the learning rate, either globally or separately for each weight, based in some heuristic way on the history of the computation. The momentum term used in standard back-propagation is a form of this strategy; so are the fixed schedules for parameter adjustment that are recommended in [12], though in this case the adjustment is based upon the experience of the programmer rather than that of the network. Franzini [4] has investigated a technique that heuristically adjusts the global ε parameter, increasing it whenever two successive gradient vectors are nearly the same and decreasing it otherwise. Jacobs [5] has conducted an empirical study comparing standard backprop with momentum to a rule that dynamically adjusts a separate learning-rate parameter for each weight. Cater [2] uses a more complex heuristic for adjusting the learning rate. All of these methods improve the overall learning speed to some degree.

The other kind of approach makes explicit use of the second derivative of the error with respect to each weight. Given this information, we can select a new set of weights using Newton's method or some more sophisticated optimization technique. Unfortunately, it requires a very costly global computation to derive the true second derivative, so some approximation is used. Parker [8], Watrous [17], and Becker and LeCun [1] have all been active in this area. Watrous has implemented two such algorithms and tried them on the XOR problem. He claims some improvement over back-propagation, but it does not appear that his methods will scale up well to much larger problems.

I have developed an algorithm that I call "quickprop" that has some connection to both of these traditions. It is a second-order method, based loosely on Newton's method, but in spirit it is more heuristic than formal. Everything proceeds as in standard back-propagation, but for each weight I keep a copy of the $\partial E/\partial w(t-1)$, the error derivative computed during the previous training epoch, along with the difference between the current and previous values of this weight. The $\partial E/\partial w(t)$ value for the current training epoch is also available at weight-update time.

I then make two risky assumptions: first, that the error vs. weight curve for each weight can be approximated by a parabola whose arms open upward; second, that the change in the slope of the error curve, as seen by each weight, is not affected by all the other weights that are changing at the same time. For each weight, independently, we use the previous and current error slopes and the weight-change between the points at which these slopes were measured to determine a parabola; we then jump directly to the minimum point of this parabola. The computation is very simple, and it uses only the information local to the weight being updated:

$$\Delta w(t) = \frac{S(t)}{S(t-1) - S(t)} \Delta w(t-1)$$

where $S(t)$ and $S(t-1)$ are the current and previous values of $\partial E/\partial w$. Of course, this new value is only a crude approximation to the optimum value for the weight, but when applied iteratively this method is surprisingly effective. Notice that the old α parameter is gone, though we will need to keep ε (see below).

Using this update formula, if the current slope is somewhat smaller than the previous one, but in the same direction, the weight will change again in the same direction. The step may be large or small, depending on how much the slope was reduced by the previous step. If the current slope is in the opposite direction from the previous one, that means that we have crossed over the minimum and that we are now on the opposite side of the valley. In this case, the next step will place us somewhere between the current and previous positions. The third case occurs when the current slope is in the same direction as the previous slope, but is the same size or larger in magnitude. If we were to blindly follow the formula in this case, we would end up taking an infinite step or actually moving backwards, up the current slope and toward a local maximum.

I have experimented with several ways of handling this third situation. The method that seems to work best is to create a new parameter, which I call μ, the "maximum growth factor". No weight step is allowed to be greater in magnitude than μ times the previous step for that weight; if the step computed by the quickprop formula would be too large, infinite, or uphill on the current slope, we instead use μ times the previous step as the size of the new step. The idea is that if, instead of flattening out, the error curve actually becomes steeper as you move down it, you can afford to accelerate, but within limits. Since there is some "noise" coming from the simultaneous update of other units, we don't want to extrapolate too far from a finite baseline. Experiments show that if μ is too large, the network behaves chaotically and fails to converge. The optimal value of μ depends to some extent upon the type of problem, but a value of 1.75 works well for a wide range of problems.

Since quickprop changes weights based on what happened during the previous weight update, we need some way to bootstrap the process. In addition, we need a way to restart the learning process for a weight that has previously taken a step of size zero but that now is seeing a non-zero-slope because something has changed elsewhere in the network. The obvious move is to use gradient descent, based on the current slope and some learning rate ε, to start the process and to restart the process for any weight that has a previous step size of zero.

It took me several tries to get this "ignition" process working well. Originally I picked a small threshold and switched from the quadratic approximation to gradient descent whenever the previous weight fell below this threshold. This worked fairly well, but I came to suspect that odd things were happening in the vicinity of the threshold, especially for very large encoder problems. I replaced this mechanism with one that always added a gradient descent term to the step computed by the quadratic method. This worked well when a weight was moving down a slope, but it led to oscillation when the weight overshot the minimum and had to come back: the quadratic method would accurately locate the bottom of the parabola, and the gradient descent term would then push the weight past this point.

My current version of quickprop always adds ε times the current slope to the Δw value computed by the quadratic formula, unless the current slope is opposite in sign from the previous slope; in that case, the quadratic term is used alone.

One final refinement is required. For some problems, quickprop will allow some of the weights to grow very large. This leads to floating-point overflow errors in the middle of a training session. I fix this by adding a small weight-decay term to the slope computed for each weight. This keeps the weights within an acceptable range.

Quickprop can suffer from the same "flat spot" problems as standard backprop, so I always run it with the sigmoid-prime function modified by the addition of 0.1, as described in the previous section.

With the normal linear error function, the following result was the best one obtained using quickprop:

Problem	Trials	ε	μ	r	Max	Min	Average	S. D.
10-5-10	100	1.5	1.75	2.0	72	13	22.1	8.9

With the addition of the hyperbolic arctan error function, quickprop did better still:

Presented at SCC-95

http://www.docs.uu.se/docs/ann/papers.html

Some Experiments Using Extra Output Learning to Hint Multi Layer Perceptrons

Olle Gällmo and Jakob Carlström

Department of Computer Systems,
Uppsala University,
Box 325, S-751 05 Uppsala, Sweden
Fax: +46 18 55 02 25, URL: http://www.docs.uu.se,
Email: Olle.Gallmo@docs.uu.se, Jakob.Carlstrom@docs.uu.se

Abstract

This paper discusses a method to exploit prior knowledge when training a neural network, called Extra Output Learning. This method makes it possible to supply hints to the network during training through extra outputs, which can later be removed. No changes to the learning algorithm or the error criterion are needed, and the method is therefore easy to apply. The effects these extra outputs have on the network are explained and tested on simple classification and function approximation problems. The method is also tested on a more realistic problem: Link Admission Control in ATM telecommunication networks.

1 Introduction

One of the reasons behind the exponential growth of interest in artificial neural networks the last decade, may be that they are trained by giving examples. In order to teach a neural network to solve a problem, the teacher does not necessarily have to know how to solve it herself, she only has to supply the data.

Of course, in practice the teacher always has at least some prior knowledge on the problem, and it should be possible to exploit this knowledge when training a neural network. Many methods to do this can be found in the literature [AR91, DGS93, OG92, ZDS92], though most of them are only useful in a narrow application field. A very general method, on the other hand, is one proposed by Suddarth and others [SSH88, YS90] called 'injection of hints' or 'extra output learning'.

The extra output learning method can be viewed as a preprocessing method, though perhaps not in the meaning normally associated to the word. The importance of suitable preprocessing in real world applications can not be over-stressed. Preprocessing is the most powerful method for incorporating prior knowledge in a neural network application, and should not (as is perhaps too often the case) be taken lightly.

Usually, the reason for preprocessing is to reduce the amount of information in the raw input data. Finding an appropriate preprocessing algorithm can be a difficult task, since the designer has to identify relevant features to keep and redundant features to discard. In other words, designing a suitable preprocessing algorithm is to exploit prior knowledge about the data.

Why not, then, use preprocessing to *add* information instead to help the neural network solve the problem? A trivial example is adding an extra input to a XOR network which is 1 if both of the other two inputs are 1. In this case, the preprocessing actually makes the problem linearly separable and it can therefore be solved by a single neuron.

The drawback is that this preprocessing will have to take place also after training, i.e. the extra inputs must be supplied also when the network is tested or used. The method may therefore have a very bad effect on the network response time, especially if the required extra input information is hard to obtain or computationally heavy to calculate.

The conclusion is that the hinting information should only be necessary to apply during training. Adding extra computations during the operational phase should be avoided.

2 Extra output learning

One way to do this is to preprocess the *target* values, i.e. supply the network with extra outputs to be trained on targets which express some knowledge about the problem [SSH88, YS90]. The point is that the extra outputs can be removed after training. But perhaps the biggest advantage for real world applications is that, unlike most other hint methods, no modification of the network algorithm or error criterion is involved. Examples of hints which can be supplied this way are knowledge on useful intermediate results [SSH88] and on monotonic regions in the target function [BH90].

It should be pointed out that the Multi Layer Perceptron (MLP) is a universal function approximator in the sense that, given a sufficient number of hidden nodes, it can *represent* any function to any degree of accuracy [HSW89]. This is not the same thing as being able to *learn* the function, however. The back propagation algorithm is a gradient descent algorithm and can, as such, get caught in local minima. Extra Output Learning can not increase the representational power of the network, but it can increase the probability of finding a good representation.

Training a neural network on a problem is a function approximation task. The goal is to train the network to implement a function, **F**, but complete information on this function is seldom available. The network is only trained on a finite and more or less randomly generated training set, **S**.

Let $\{\overline{\mathbf{F}}\}$ denote the set of functions the network can implement to satisfy **S**. Here, 'to satisfy a set' means that the success criterion is met for the **S**-set, for example that the mean squared error is less than a certain limit or that all input vectors are classified correctly. In other words, the functions in $\{\overline{\mathbf{F}}\}$ are models of **F**.

The set $\{\overline{\mathbf{F}}\}$ set can be quite large, but some of its members will not be good models of **F**, since the training set is finite. These members correspond to points in the error space which appear to be global minima in the limited resolution of the finite training set, but which in reality is not.

Now, make another training set **S'** by applying a function **H** on the same input vectors as in **S**. Let $\{\overline{\mathbf{H}}\}$ denote the set of functions which the network can implement to satisfy **S'**. Since **S** and **S'** contain the same input vectors, a network can be trained to approximate both functions simultaneously, by extending the output layer to represent the output vectors of both **S** and **S'**. This corresponds to two networks sharing the same hidden nodes, trained on the two functions.

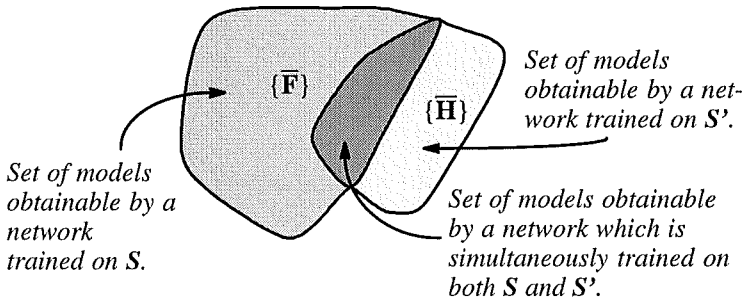


Figure 1: Model reduction through extra output learning.

The network is now forced to find an internal representation which satisfies both training sets, which means that the set of models is reduced to $\{\overline{\mathbf{F}}\} \cap \{\overline{\mathbf{H}}\}$ (see *Figure 1*). Perfect generalization is obtained when a function is found which is a sufficiently good model of the target function, **F**. The goal is to find a hint **H** which cuts away more bad models than good ones, thus yielding a higher probability of finding a good model in the intersection. If such a hint is found, the network will learn faster (fewer models to choose from) and generalize better (higher probability of finding a good model).

A poor choice of a hint, however, may even reduce generalization ability.

- The hint function and the target function must be correlated, in the sense that they must have a common sub-function for the hidden nodes to find. If there is no such sub-function, the hint is useless.
- Increasing the complexity of the functional relationship between the inputs and the outputs may take the network to a point where it simply cannot represent both functions due to a lack of hidden nodes. This corresponds to the two sets in *Figure 1* being disjoint.

The obvious conclusion is that hints should be relevant to the problem at hand and not too complex. A discussion on the probability of a hint increasing the generalization ability of the network can be found in [SK90].

3 Classification experiments

A simple example of the effect extra output learning can have is the famous XOR problem. A 2-2-1 MLP was trained on this problem until all four input vectors were classified correctly (where an output value <0.4 was treated as a 0, a value >0.6 as a 1 and all values in between as unsure, i.e. wrong). If perfect classification had not been obtained within 1000 epochs, that session was considered to have failed. Failed sessions were counted but not included in the averages and standard deviation figures reported below.

On average, over 100 training sessions with different initial weights, the conventional network solved the problem in 350 epochs with a standard deviation of 158 epochs (*Figure 2a*). The network failed to solve the problem on 4 occasions.

For the same set of 100 initial weight settings as before, the network was now trained using AND as an extra target. This should be a relevant hint since XOR can be defined in terms of it. The network failed on 2 occasions and, on average, solved the problem in 155 epochs with a standard deviation of 59 epochs (*Figure 2b*).

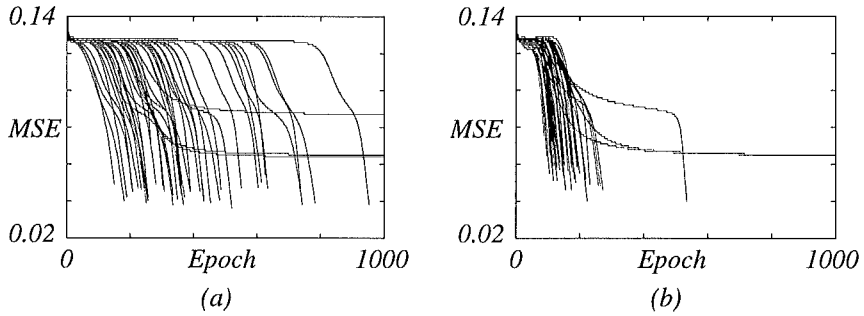


Figure 2: History of the Mean Squared Error of the XOR output for 100 sessions of the conventional network (a) and the hinted network (b). The initial state for each session was the same in both cases.

So, three things can be noted about extra output learning on the XOR problem:

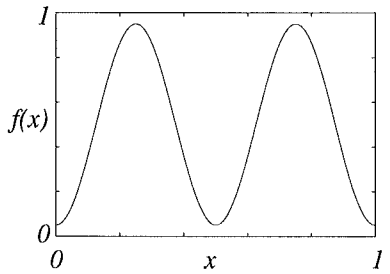
- Training is faster by more than a factor two, on average. The hinted network was actually faster in all converging cases, though on two occasions the gain was as small as a factor 1.17.
- The training time is more deterministic, i.e. the number of paths to the goal has decreased. This suggests that the error surface is more smooth.
- The probability of getting stuck in a local minima is lower in the hinted network. This suggests better generalization, though the experiment says nothing of this issue directly since the training set is complete.

For a more thorough analysis of the effects extra output learning has on networks solving the XOR problem, see [SK90].

4 Function approximation experiments

XOR is a classification task. What about continuous function approximation? [BH90] propose that extra output learning can be used to hint the network about monotonic regions in the target function.

To test this, we generated a training set of 100 input-output pairs and a test set of 1000 pairs, using the target function in (*Figure 3*). The x-values in both sets were evenly distributed in the interval [0, 1[where there are four monotonic regions.



$$f(x) = \frac{10 - 9cos(4\pi x)}{20}$$

Figure 3: The target function with its four monotonic regions.

First, a conventional 1-5-1 network was trained on this set 10 times from different starting points in weight space. The result on the test set after 2000 epochs can be seen in *Figure 4a*. In eight of the ten cases, the network got stuck in local minima where the curve follows the target for a while but then flattens out to the right. That these cases are local minima is indicated by the mean squared error history in *Figure 4b*. They cannot be explained as a shortage of representational power (hidden nodes) either, since in two cases the network did manage to approximate the target function very well.

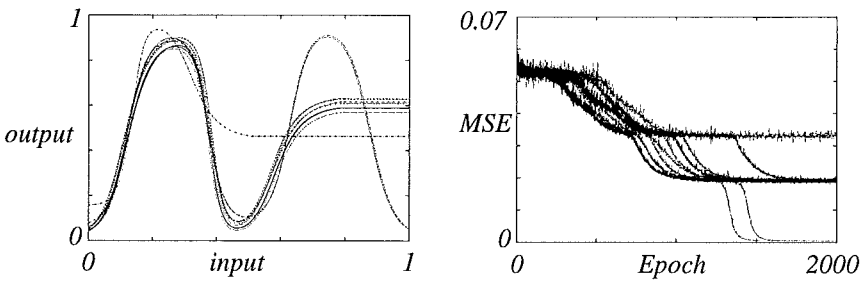


Figure 4: The conventional network's response on the test set (a) and the mean squared error history on the training set (b).

One way to do better is to add more training data and/or hidden nodes. Another way is to hint the network about the four monotonic regions. Hopefully, this information will convince the network early on that the flat parts of the curves in *Figure 4a* are bad.

The idea is to add one extra output for each monotonic region, and train these on targets $g_i(x)$ which are equal to $f(x)$ within the region but constant outside it (*Figure 5*).

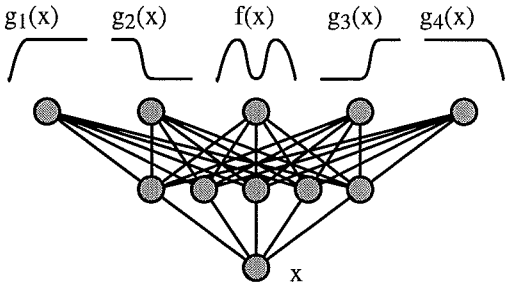


Figure 5: Using monotonic regions as hints. The $g_i(x)$ functions are help functions representing monotonic regions in the actual target function $f(x)$. Each help function is equal to f within the corresponding region and constant outside it.

The hinted network, trained from the same starting point in weight space as the conventional network, learned much faster (*Figure 6b*) and never found a malicious local minimum. In other words, the expected result of training such a net is more deterministic, as is shown by the 10 approximations in *Figure 6a*.

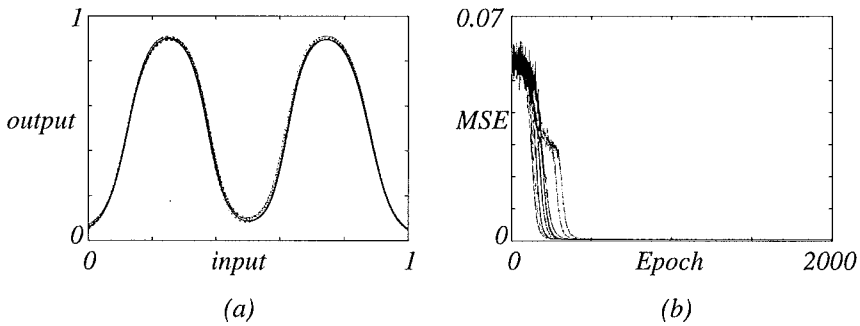


Figure 6: The hinted network's response on the test set (a) and the mean squared error history on the training set (b).

There is often a wide gap between the number of hidden nodes which are required of an MLP to *represent* a function, and the number of hidden nodes required *in practice* to sufficiently reduce the risk of falling into malicious local minima. Extra output learning offers a way to reduce this risk and it can therefore be said to narrow the gap between the theoretical and practical requirements on the number of hidden nodes. This, in turn, suggests that the hinted network should require less training data than a network without hints [BH90], though this issue is beyond the scope of this paper.

5 A more realistic example: ATM Link Admission Control

ATM (Asynchronous Transfer Mode) is a connection oriented, packet switched, transport mode for broadband telecommunication networks [HH91]. The idea is to support virtually all types of communication services on the same network by asynchronous multiplexing of fixed size packets, called cells.

Small buffers on each outgoing link from a switch take care of simultaneously arriving cells. If a buffer saturates subsequent cells may be lost. The probability of cell loss (P_{loss}) is therefore an important quality measure in the network, and it is up to each link to decide if a new connection can be admitted or not without exceeding the highest acceptable cell loss probability (here 10^{-9}). This preventive decision procedure is called *Link Admission Control* (LAC).

Each connection is characterized by three parameters supplied by the user on connection setup. Accurate fluid-flow procedures to estimate the cell loss probability on a link, given the parameters for each connection, do exist. However, they are too complex to be used in real time. One solution to this problem is to train a Multi Layer Perceptron to model the accurate procedure [NGA+92, GNGA93, NGGA93]. Also hybrid solutions, where a neural network and approximations of the exact procedures are combined, have been considered [Nord93, BNG+95].

The connection parameters of all connections already sharing the link can not be given directly to the neural network, since it would require a dynamic size of the input vector. Furthermore, the cell loss probability estimate should be independent of the order in which the connections were admitted. Therefore, the connection parameters are aggregated to some permutationally invariant state vector. Here, the 6-component state vector proposed in [NGGA93] is used.

A large set of over 100,000 random traffic situations has been generated. Each traffic situation consists of a state vector for the traffic already sharing the link, the new connection request and the corresponding P_{loss} estimate, as calculated by the accurate fluid-flow procedure. Note that this is not the same set as in our previous work [NGA+92, GNGA93, NGGA93]. The new set is larger and considered to be closer to reality. A more detailed description of the new set can be found in [BNG+95]. In the experiments described below, a subset of 500 situations is used for training and another subset of 5000 situations for testing.

Four experiments are described below. First, a conventional MLP without hints is applied to the admission control problem. Then, three different attempts to find extra output hints are described. A summary of the results from these experiments can be found in *Table 1* at the end of this section.

5.1 Training a neural network without hints

As a reference to evaluate the hint experiments, a network with 9 inputs (6 for the state vector and 3 for the connection parameters), 5 hidden nodes and 1 linear output was trained for 1000 epochs on the training set. This was repeated 10 times for different initial weight settings. The result on the test set was a hit rate (relative number of correct decisions) of 88.5% on average, with an absolute standard deviation σ=2.0%.

But the hit rate is not the only measure of success. The effects of the bad decisions should also be considered [GNGA93]. For example, accepting a connection with a cell loss probability of 10^{-8}, when the highest acceptable probability is 10^{-9}, is not as serious as accepting one with probability 10^{-5}. It is therefore useful to look at how far from the acceptable probability limit the network makes mistakes.

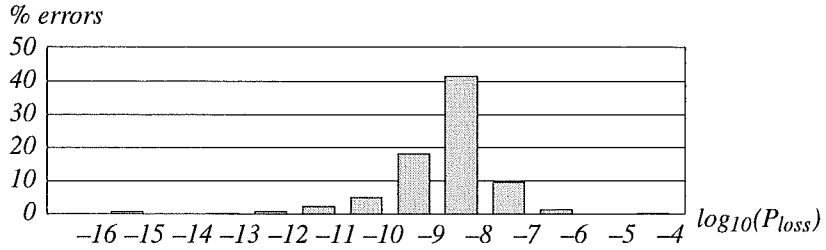


Figure 7: Distribution of errors for the conventional neural ATM admission controller. The length of each bar corresponds to the number of decision errors within a target interval, as a percentage of the total number of situations in the same interval. Errors above -9 are potentially bad for the customers and errors below -9 lead to bad utilization.

The histogram in (*Figure 7*) shows how the decision errors for the 10 runs were distributed. Note that the network classifies over 40% of the situations in the interval $[10^{-9}, 10^{-8}]$ incorrectly. This is not as serious as it may seem, however, since the effect of these mistakes are small. The goal should be to reduce the number of bad decisions without increasing the number of mistakes far from the 10^{-9} decision border.

To evaluate this, something is needed to measure the 'width' of the histogram. Here, the width, denoted ω, is defined as the mean squared distance from the decision border for all traffic situations which lead to bad decisions:

$$\omega = \frac{\sum_{\forall errors} (log_{10}(P_{loss}) - log_{10}(10^{-9}))^2}{N_{errors}} = \frac{\sum_{\forall errors} (log_{10}(P_{loss}) + 9)^2}{N_{errors}}$$

where N_{errors} is the total number of bad decisions. In this experiment, ω=0.66.

5.2 Hinting with current P_{loss}

In [SSH88] it is suggested that the extra output method can be used to back propagate intermediate results which the teacher knows are useful when solving the problem. For ATM Admission Control, an intuitively useful intermediate result is the cell loss probability on the link before the decision, i.e. excluding the new connection.

However, it turned out that hinting the network with this figure did no good at all. 10 runs with the same initial conditions as the conventional network described above, resulted in an average hit rate of 87.1% (σ=2.1%). The error distribution was not very good either (ω=0.87). The hinted network did better than the conventional network in the interval $[10^{-9}, 10^{-8}]$, but did worse in all other intervals. In

particular, the hinted network introduced an error in the interval $[10^{-3}, 10^{-2}]$, which could have very serious effects on the quality of service in a real ATM network.

A possible explanation is that the hint is *too* correlated to the original target output. Speaking in terms of the two model sets in (*Figure 1*): If the two sets are about the same size and the overlap is substantial, there will be almost no reduction in the number of models. Therefore, adding such a hint will not help the network and may even confuse it, as it did here.

5.3 Moving a complex input to the outputs

The 6'th component (c_6) in the state vector has turned out to be important for the network; Removing it from the input vector reduces the average hit rate to 82.9% (σ=0.9%) and the error distribution (ω=2.5) is terrible. However, this component is comparatively hard to calculate and it would therefore be nice if it only had to be supplied during training.

Unfortunately, hinting with c_6 as an extra output did not work either. The hit rate was 82.6% (σ=1.3%). Though the width (ω=2.3) is slightly better than a network where the component has been removed completely, it is still far too wide. In section 2 it was pointed out that the hint should not be too complex. Here it is even worse: c_6 is not a function of the other 5 components in the state vector. Therefore, using it as a hint only confuses the network.

5.4 Hinting with a sub-interval of the actual target

Though the neural network is trained as a continuous function approximator, its estimate of the cell loss probability is intended for decision making, i.e. saying "yes" or "no" to the new connection. The main reason why the network is trained as a function approximator and not as a classifier, is that its estimate of the cell loss probability is useful in other traffic control procedures in the ATM network. However, giving the network a hint that its output is to be used for classification purposes might be a good idea. Such a hint should indicate to the network that small errors in the P_{loss} estimate are more serious near the border than far from it, since a small estimation error far from the border is unlikely to change the decision. (*Decision* errors, however, are more serious far from the border than near it.)

In section 4, a function approximation network was hinted about important (monotonic) regions in the target function, by adding extra outputs corresponding to these regions. For the admission controller network, the important region is 'near the decision border', say within two magnitudes from it. So, in the same spirit as in the monotonic region example, the neural admission controller was supplemented with an extra output, trained to be equal to the the target P_{loss} figure within the interval $[10^{-11}, 10^{-7}]$ and constant outside it.

10 training sessions from the same 10 initial conditions as the network without hints, resulted in a hit rate of 90.3% (σ=0.5%). This is a significant improvement, both in terms of hit rate and in standard deviation. The latter confirms what was mentioned earlier, that a good hint should make training more deterministic in its outcome.

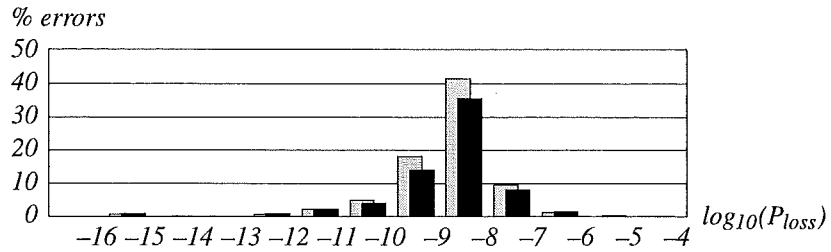


Figure 8: Distribution of errors for the network hinted with the region $[10^{-11}, 10^{-7}]$ (black), compared to the network without hints (gray).

The error distribution can be seen in (*Figure 8*). Within the region where the network got extra help $[10^{-11}, 10^{-7}]$, the hinted network is consistently better. In the outer regions, on the other hand, the network without hints is consistently equal to, or even slightly better than, the hinted network. This means that the width is slightly worse for the hinted network (ω=0.74), but it is not as wide as in the other hint experiments described above.

The conclusion is that extra output learning can be used to shift the attention of the network to certain areas of interest. In [GNGA93] other ways to shift the attention of a neural network are discussed, also based on preprocessing of the training set but not using extra outputs.

Table 1 summarizes the results of the experiments in this section. For comparison, the best known approximation to the exact procedure (the U_{LD} approximation defined in [BNG+95]) yields a hit rate of 82.3% and a width of 0.59 on the same test set. The lower width is due to the fact that all the decision errors are for traffic situations in the interval $[10^{-12}, 10^{-9}]$ which is very good. On the other hand, it classifies 89.5% of the situations in the interval $[10^{-10}, 10^{-9}]$ incorrectly.

Network	**Hit rate (correct decisions)**	**Absolute standard deviation σ**	**Width ω**
without hints	88.5 %	2.0 %	0.66
with current P_{loss} hint	87.1 %	2.1 %	0.87
with 6'th component hint	82.6 %	1.3 %	2.30
with region hint	90.3 %	0.5 %	0.74

Table 1: Summary of results of the Admission Control experiments.

6 Conclusion

The extra output learning method consists of finding a good hint which is a function from the same input domain as the target function. The hint function is given as an extra target during training. This forces the hidden layer to find a representation which makes it possible for the network to 'satisfy' both targets. Thus, the number of models the network can form of its original target is reduced.

The main difficulty is to find good hints. What seems like a relevant hint is not necessarily a good one, as was shown in section 5.2. Once a good hint has been found, however, the extra output learning method has several advantages:

- It is easy to use, since it requires no modification of the training algorithm, nor any permanent modification of the network structure.
- Extra information and corresponding modifications to the structure of the network are only needed during training. No extra work is required in the recall phase since the extra outputs are discarded after training.
- The method reduces the risk of getting caught in local minima during training, though it cannot eliminate the risk completely. Since also the hinted network is trained by a gradient descent algorithm, there is always a risk of getting stuck.
- The experiments on simple classification and function approximation show that the gain in speed can be substantial.

In ATM Link Admission Control, the best results have been obtained when the network was given the critical region near 10^{-9} as a hint, where even small errors in the cell loss probability estimate can lead to false decisions. The result is an admission controller with a lower error rate, but the bad effects these errors may have on the link are slightly greater. An optimal hint would reduce both the error rate and the effects of the few errors made, but such a hint has yet to be found.

Acknowledgements

The authors would like to thank Lars Asplund and Ernst Nordström for their ideas and comments on earlier versions of this paper.

This work was financially supported by ELLEMTEL Telecommunication Systems Laboratories and by NUTEK, the Swedish National Board for Industrial and Technical Development, which is also greatly acknowledged.

References

[AR91] K.A. Al-Mashouq & I.S. Reed, Including hints in training neural networks,*Neural Computation*, **vol. 3**, **no. 3**, pp. 418-427, 1991.

[BH90] A.B. Baruah & A.D.C. Holden, Back Propagation and Monotonic Functions, *Proceedings of the International Neural Network Conference*, **vol. 1**. pp. 383-386, Paris, France, 1990.

[BNG+95] H. Brandt, E. Nordström, O. Gällmo, M. Gustafsson & L. Asplund, A Hybrid Neural Network Approach to ATM Admission Control, to appear in *Proceedings of the International Switching Symposium (ISS'95)*, p. P.b6, Berlin, April 1995.

[DGS93] S. Das & C.L. Giles & G.-Z. Sun, Using Prior Knowledge in an NNPDA to Learn Context-Free Languages, in C.L. Giles, S.J. Hanson & J.D. Cowan (Eds.), *Advances in Neural Information Processing Systems* **5**, pp. 65-72, Morgan Kaufmann, 1993.

[GNGA93] O. Gällmo, E. Nordström, M. Gustafsson & L. Asplund, Neural Networks for Preventive Traffic Control in Broadband ATM Networks, *International Workshop on Mechantronical Systems for Perception and Action (MCPA'93)*, pp. 139-145, Halmstad, Sweden, 1993.

[HH91] R. Händel & M. N. Huber, *Integrated Broadband Networks: An Introduction to ATM-based Networks*, Addison-Wesley, 1991.

[HSW89] K. Hornik, M. Stinchcombe & H. White, Multilayer Feedforward Networks Are Universal Approximators, *Neural networks* **2**, pp. 359-366, 1989.

[Nord93] E. Nordström, A Hybrid Admission Control Scheme for Broadband ATM Traffic, in J. Alspector, R. Goodman & T. X. Brown (Eds.), *Proceedings of the International Workshop on Applications of Neural Networks to Telecommunications (IWANNT*93)*, pp. 77-84, Lawrence Erlbaum, 1993.

[NGA+92] E. Nordström, O. Gällmo, L. Asplund, M. Gustafsson & B. Eriksson, Neural Networks for Admission Control in an ATM Network, in L.F. Niklasson & M.B. Bodén (Eds.), *Connectionism in a Broad Perspective: Selected papers from the Swedish Conference on Connectionism - 1992*, pp. 239-250, 1994.

[NGGA93] E. Nordström, O. Gällmo, M. Gustafsson & L. Asplund, Statistical Preprocessing for Service Quality Estimation in a Broadband Network, *World Congress on Neural Networks WCNN'93*, **vol. 1**, pp. 295-299, Portland, Oregon, 1993.

[OG92] C.W. Omlin & C.L. Giles, Training Second-Order Recurrent Neural Networks using Hints, *Machine Learning: Proceedings of the Ninth International Conference (ML92)*, D. Sleeman & P. Edwards (Eds.), Morgan Kaufmann, p. 363, 1992.

[SK90] S.C. Suddarth & Y.L. Kergosien, Rule-Injection Hints as a Means of Improving Network Performance and Learning Time, in L.B. Almeida & C.J. Wellekens (Eds.), *Neural Networks*, Lecture Notes in Computer Science, **vol. 412**, pp. 120-129, Springer Verlag, 1990.

[SSH88] S.C. Suddarth, S.A. Sutton & A.D.C. Holden, A Symbolic-Neural Method for Solving Control Problems, *1988 IEEE International Conference on Neural Networks*, **vol. 1**, pp. 515-523, San Diego, CA, 1988.

[YS90] Y.-H. Yu & R.F. Simmons, Extra Output Biased Learning, *Proceedings of the International Joint Conference on Neural Networks (IJCNN-90)*, **vol. 3**, pp. 161-166, San Diego, CA, 1990.

[ZDS92] S. Zhao, T.S. Dillon & S. Sestito, Development of Multilayer Neural networks in the Presence of Constraints, *Proceedings of the International Joint Conference on Neural Networks (IJCNN-92)*, **vol. 2**, pp. 462-467, Beijing, China, 1992.