Complex Systems **6** (1992) 479–494

Refined Pruning Techniques for Feed-forward Neural Networks

Anthony N. Burkitt*
Computer Sciences Laboratory, Research School of Physical Sciences, Australian National University, GPO Box 4, Canberra, ACT 2601, Australia

Peer Ueberholz
Physics Department, University of Wuppertal, Gauss-Straße 20, D-5600 Wuppertal 1, Germany

Abstract. Pruning algorithms for feed-forward neural networks typically have the undesirable side effect of interfering with the learning procedure. The network reduction algorithm presented in this paper is implemented by considering only directions in weight space that are orthogonal to those required by the learning algorithm. In this way, the network reduction algorithm chooses a minimal network from among the set of networks with constant E-function values. It thus avoids introducing any inconsistency with learning by explicitly using the redundancy inherent in an oversize network. The method is tested on boolean problems and shown to be very useful in practice.

1. Introduction

The problem of choosing the optimal network size and architecture for a given situation has remained intractable. Once a network architecture has been decided upon, learning procedures such as back-propagation [1] enable the network to be trained, provided that there are enough degrees of freedom (hidden units, weights and biases). One of the principal methods for solving the network size problem in practice is pruning an oversize network; that is, eliminating some of the weights according to some appropriate criteria. A variety of such algorithms have been described in the literature [2, 3, 4, 5, 6, 7, 8].

The primary reason for the extensive interest in such network optimization algorithms is the fact that minimal nets tend to have the best generalization properties [9, 10], as demonstrated both by recent rigorousresults [11, 12] and empirical investigations [4]. Networks that are too large are able to fit

*Electronic mail address: tony@nimbus.anu.edu.au

the training data arbitrarily closely but, in essence, they memorize the training patterns, and typically do not generalize well to new inputs. Another motivating fact is that the amount of computation both for forward computation and for learning grows with increasing network size. We hope to gain some insight into the behavior of a minimal network, in terms of a set of rules, or as a function of the hidden units.

A number of network reduction schemes have been proposed that adds terms to the standard sum squared error cost function,

$$E_{\text{err}} = \sum_n E_p = \frac{1}{2} \sum_{n,p} (t_{pn} - o_{pn})^2 \tag{1}$$

where the sum is over the n output units and p input/output training patterns, whose target output and network output on the output unit n are t_{pn} and o_{pn}, respectively. In the method presented in this paper, an oversized network is taken for the initial network configuration, and is trained using back-propagation in the usual way. Training can be considered as a minimization procedure for the function E_{err} in the space of weights (and biases). Because the network is oversized, there will be many network configurations that correspond to the same value of E_{err}. Such equivalent network configurations can be visualized as *contours* on the surface of the cost function—in the same way that contours on a topographical map define points of equal height. The method we present here can be viewed as a way of moving around these "contours" to seek a configuration where as many as possible of the weights vanish. The method is "orthogonal" to learning in the sense that the pruning algorithm moves the network configuration around the contours of the cost function, which are perpendicular in weight space to the gradient of the cost function, which in turn is required for learning. The details of this method are presented in the next section.

Other pruning algorithms typically interfere quite severely with network learning. The standard way to prune a network is to introduce additional terms to the cost function (1), and a number of such schemes have been discussed in the literature [2, 3, 4, 5, 6, 7]. One of the simplest such cost functions is the sum over all weights in the network, with each weight providing a quadratic contribution. This method, also called *weight decay* [2, 3, 4], delivers at each iteration step a contribution that decreases every weight by a certain uniform proportion; that is, large and small weights decay at the same rate. The weights which do not contribute to learning will then have an exponential time decay. A number of more complex weight cost functions have been introduced in order to influence weights that lie within specific ranges. One such method is *weight elimination* [5], where the additional term can be thought of as a complexity term that likewise encourages the reduction and eventual elimination of many of the connections. The essential problem with such methods is that the additional terms in the cost function interfere with learning, since they change the shape of the contours and the positions of the minima of the error cost function. It is necessary to carefully adjust the strength of the additional terms so that network reduction

is achieved, yet the network still converges to a training solution. Because these goals are not necessarily compatible, the success rate of convergence typically decreases—often dramatically. The method of Karnin [6] has some similarity with the method that we propose, in that learning is carried out using only the quadratic error function (1). Superfluous weights are eliminated by estimating the sensitivity of the global cost function to each of the weights. This can be accomplished by keeping track of the incremental changes to the weights during learning, and by discarding connections with weights that have a low sensitivity. Karnin's method has the advantage that no modification of the cost function is required, and there is therefore no interference with the learning process while the sensitivities are being measured. However, the method is not dynamic—it is not able to move the network configuration to other potentially more favorable regions of weight space in some systematic manner.

Our method is presented and explained in the following section, together with several possible variations and some details about its implementation. Section 3 contains results of the method as applied to the parity and symmetry problems with a single output unit and one hidden layer.

2. The method: pruning orthogonally to learning

Oversized networks have an inherent redundancy; it is possible to find a network of smaller size (i.e., fewer weights and biases) that is still capable of learning the given tasks. This inherent redundancy means that any particular value of the cost function (1) has a large number of possible network configurations associated with it. Furthermore, if we have one such network configuration, then there is a continuous set of weights and biases that have the same value of the cost function. This set of weights defines a contour of the cost function in the space of all possible weights. Therefore, if we find a network configuration with a cost function value of $E_{\text{err}} = \xi$ (by using back-propagation or any other training method, for example), then it is possible to find a large set of equivalent network configurations by moving around such contours in weight space. Since we also wish our network configurations to have good generalization properties, it is reasonable to use this freedom of choice to seek a network configuration in which as many weights as possible vanish. Learning proceeds with a minimization in the weight space of the error-squared cost function E_{err} (1)—*without* adding any additional terms—that presents all the patterns to be learned. (We refer here to batch learning only.) A standard training procedure is used—such as back-propagation [1] or any of the more recent accelerated variations and improvements [13, 14, 15, 16, 17]. The network reduction proceeds by keeping E_{err} constant. We require the gradient of E_{err} in weight space for learning, which is exactly orthogonal to the contour $E_{\text{err}} = \text{constant}$. This provides a clear conceptual separation of the network learning and network reduction procedures.

In order to move around the contours E_{err} = constant we introduce the function E_{nr} for the network reduction algorithm. There are a number of possibilities for this function that we will discuss, some of which have been introduced in the literature in the context of other network pruning algorithms. In our notation, a network configuration is represented by a vector $\vec{w}$ in the space of weights and biases. We likewise define the gradient of E_{err} by

$$\vec{\Omega}_{\text{err}} = -\frac{\partial E_{\text{err}}}{\partial \vec{w}} \tag{2}$$

and similarly the gradient of the network reduction term by

$$\vec{\Omega}^1_{\text{nr}} = -\frac{\partial E_{\text{nr}}}{\partial \vec{w}} \tag{3}$$

However, in order to ensure that we remain on a contour of constant E_{err} it is necessary to add a *correction* term to $\vec{\Omega}^1_{\text{nr}}$ of the following form:

$$\vec{\Omega}_{\text{nr}} = \vec{\Omega}^1_{\text{nr}} - \beta \vec{\Omega}_{\text{err}} \tag{4}$$

where β is given by

$$\beta = \vec{\Omega}_{\text{err}} \cdot \vec{\Omega}^1_{\text{nr}} \Big/ \vec{\Omega}_{\text{err}} \cdot \vec{\Omega}_{\text{err}} \tag{5}$$

and where the vector product does *not* include the contributions from biases—for reasons that will be discussed later in this section. By moving through weight space in the direction of $\vec{\Omega}_{\text{nr}}$ we generate a series of configurations with the same E_{err} value (apart from small deviations introduced by the finite step-size error associated with discrete changes in the weights).

By an appropriate choice of the function E_{nr} it is possible to reduce redundant weights towards zero. The algorithm begins with an oversized random initial network configuration and implements back-propagation, including contributions from both $\vec{\Omega}_{\text{err}}$ and $\vec{\Omega}_{\text{nr}}$. Weights close to zero are eliminated *only* after the network has found a first learning solution, and some criterion for eliminating small weights needs to be decided upon. Once learning has been achieved, weights whose absolute values are less than a certain fraction $f_{\min}$ of the maximum weight $w_{\max}$ in the network are set explicitly to zero, and thus eliminated from the network (we typically chose a value of $f_{\min} = 0.1$). The β term in equation (4) ensures that those weights that are essential for learning are not affected by the network reduction step.

Hidden units and biases

There are two ways in which a hidden unit can be "turned-off"; that is, eliminated from the network without affecting network performance. The first and most straightforward case occurs when the weight w_{ok} between the output unit o and the hidden unit k tends to zero (we consider the case where the network has only one output unit—generalization to the case with

a number of output units is straightforward). In this situation the hidden unit k simply can be discarded. The second case occurs when all the weights w_{ki} between a hidden unit k and the input units i_n tend to zero. The output of unit k is then the same for every input pattern p, and it is determined *only* by the bias θ_k on that unit, as follows:

$$\text{output on unit } k = c_k = (1 + e^{-\theta_k})^{-1} \tag{6}$$

for units with a sigmoid response function. A similar situation obtains when the bias becomes very large in comparison to the other weights attached to the unit. It is then possible to eliminate unit k and add its network contribution to the bias θ_o on the output unit:

$$\theta_o \rightarrow \theta_o + w_{ok} c_k \tag{7}$$

In this way the output bias θ_o absorbs the turned-off hidden unit k, which is then eliminated from the network (again, generalization to the case of many output units is straightforward). Although biases can be thought of as weights connected to a unit that is always *on*, we have found it useful to treat weights and biases separately because of the foregoing considerations (hence any reference to weights in this paper does *not* include biases, unless explicitly stated).

The function E_{nr}

The purpose of the function E_{nr} is to reduce the number of weights and hidden units in the network. The simplest way to implement this is to define E_{nr} to be the sum over all weights (but *not* biases) in the network of the absolute value of the weight:

$$E_{\text{nr}} = \mu_w \sum_{\text{weights}} |w_{ij}| \tag{8}$$

This particular definition is straightforward to implement because the partial derivatives required for $\vec{\Omega}_{\text{nr}}$ (see equations (2) and (3)) are simply ± 1, according to the sign of the respective weight. Among other possible definitions (some of which have been discussed in the literature [2, 3, 4, 5, 6, 7]) are

$$w_{ij}^2, \quad w_{ij}^2/(1 + w_{ij}^2), \quad 1 - e^{-|w_{ij}|}, \quad 1 - e^{-w_{ij}^2}, \quad \text{and} \quad \log(1 + w_{ij}^2) \tag{9}$$

as well as variations on these, such as dividing the weights by some normalization factor w_0. These functions will be referred to as *simple* functions, to denote the fact that each weight experiences a network reduction term that is independent of all other weights.

It is also possible to introduce slightly more complicated functions that still retain some locality, such as making the coefficients of such simple terms depend upon other weights connected to the *same* hidden unit. We consider

an example of such a term when we make the coefficients of simple terms inversely proportional to the number of active weights (that is, weights that haven't been eliminated) associated with the particular hidden unit (see Section 3). Such a refinement tends to eliminate the weights associated with hidden units that have already had weights eliminated, and thus to reduce the number of hidden units in the final network.

It is also worth noting that we are not restricted to terms that are explicit functions of the weights. In principle, we could also consider functions of the unit outputs o_k^p, where k denotes the unit and p the pattern, as in [3]. Introducing such terms requires some care regarding the effect that they have upon the resulting network configurations. Any expression involving the pattern output o_k^p whose minimum lies at $o_k^p = 0$ (for example, a simple quadratic term $(o_k^p)^2$) has the effect of pushing o_k^p to smaller and smaller values—which can only be achieved by *large* negative weights! From the previous discussion of turning-off hidden units and the role of biases, it is clear that a more appropriate term would be $(o_k^p - c_k)^2$ with c_k as defined in (6), since the minimum of this function (namely, $o_k^p = c_k$ for all patterns p) is exactly the condition required for turning-off the hidden unit k.

Practical implementation

A number of points concerning the practical implementation of this method require discussion. It is clear that the correction term in (4) is only necessary if β in (5) is negative. If β is positive, then the contribution from $E_{\rm nr}$ is not in conflict with learning and no correction is needed. Carrying out the correction ensures that the $E_{\rm nr}$ contribution *stays* on a contour of $E_{\rm err}$, whereas if β is positive and no correction is carried out, the network configuration moves to a contour of smaller $E_{\rm err}$—which is what is required during learning in any case. Only if β is negative, which means that network reduction conflicts with learning, is the correction carried out. Exactly this situation—that is, β being negative—has led to the sharp decrease in learning ability of many pruning algorithms.

We also need some guide concerning the relative sizes of the contributions from $E_{\rm err}$ and $E_{\rm nr}$. In principle we need only check that the step-size associated with the $E_{\rm nr}$ contribution is such that the network configuration stays very close to the contour. In practice this has been implemented by requiring that β remain larger than $\beta_{\rm lower}$ (we choose $\beta_{\rm lower} = -1.0$, although the results are not much affected by any value in the range -0.5 to -5.0); and when this is not already the case, the vector $\vec{\Omega}_{\rm nr}$ is scaled appropriately to fulfil this condition.

It is possible to carry out the network reduction procedure from the very beginning of the network training process, or to let the network first find a learning solution. We investigated alternatives and found, in general, that smaller networks resulted when the contribution from the network reduction algorithm was included from the start of training. On the other hand, more training steps were required to find the first network solution. This result is

not surprising, because the inclusion of the network reduction contributions from the start of training influences the extent of the network configuration space that is explored—and it is well known that the training rate and the number of degrees of freedom in a network are inversely related [18]. When there is no network reduction term during learning, the network tends to use all the available degrees of freedom rather than seeking a solution with a minimal number of weights and hidden units. This behavior probably indicates that such solutions are more numerous than the minimal solution(s), as discussed in [9], Section 10.

Our algorithm will not necessarily always give *the* minimal network configuration, since not all configurations with the same value of the cost function need be continuously connected via a contour. There may be a number of disconnected contours with the same E_{err} value, and upon which of these contours the network lands will depend on the (random) initial network configuration. However, the algorithm can, in principle, find the minimal configuration associated with the contour that the network is on.

Local version of the algorithm

A vastly simplified version of the algorithm results if we insist upon *locality*; that is, if each weight is iterated according only to information received from the units to which it is attached—it receives no information about the network apart from this. In this case each weight is iterated according to the contribution obtained from back-propagation and, *if the signs of the two terms are the same*, the contribution from E_{nr}. After learning has taken place, the network reduction can still proceed by continuing to calculate the contribution that would be provided by back-propagation (although it is not used), and comparing its sign with that of the network reduction term. This version of the algorithm does not require us to calculate β and there is therefore no global knowledge of the network configuration. Even such a simplified version of the algorithm ensures, however, that the network reduction contributions always move the network configuration to a point *inside* the contour; that is, to a configuration with a lower value of E_{err} (apart from the caveats about finite step-size effects). Results for this version of the algorithm are discussed in the next section.

3. Implementation on simple Boolean problems

In this section we present the results of testing this algorithm on a number of well known Boolean problems. Such problems have the advantage that they are relatively well understood and have been extensively investigated, both numerically [1] and analytically [19]. Although these problems are rather special in some regards, they have become recognized as standard "bench-tests" for the study and comparison of new methods. There are a number of specialized methods for teaching neural networks Boolean functions, some of which are very much faster than back-propagation methods—such

as decision trees [20, 21], sequential construction algorithms [22], geometric construction algorithms involving the partitioning of the input space by hyperplanes [23], and Queries [24]. However, the generalization of these methods to non-Boolean functions or situations involving on-line training remains problematic, and they have therefore not replaced back-propagation as a general purpose learning algorithm for neural networks. Indeed, the method of back-propagation has been extensively developed [13, 14, 15, 16, 17] and has become an extremely versatile tool. Such simple Boolean problems as parity and symmetry are particularly suitable for testing pruning algorithms because the solutions involve only one hidden layer (the case for any Boolean problem) and the minimal number of hidden units required by a network is known.

The parity function has output +1 when the number of input units N_I with +1 is odd and an output 0 otherwise. The parity network problem requires a minimal number of hidden units $N_H = N_I$ [1]. The symmetry problem, which has an output +1 if the pattern of the input units is symmetric about the center and an output of 0 otherwise, requires a minimum of two hidden units [1], irrespective of the number of input units.

The training patterns were presented in batch mode and the weights were adjusted according to the back-propagation algorithm [1], with learning rate ϵ and momentum α. The patterns were considered learned when the output was within a learning tolerance of 0.1 of the correct answer. The initial weights were chosen randomly on a scale $r = 2.5$.

A cutoff time was also introduced for the learning procedure whereby, if the training had not been completed within this number of iterations, the network was considered to be stuck in a local minimum. The performance of a neural network algorithm can be presented in a number of ways. The least ambiguous pair of performance indicators are the success rate and the average training time. The success rate is simply the percentage of training runs that gave the correct output (within the learning tolerance) for all training patterns before the cutoff time. For the average training time τ we use the definition [18]

$$\tau = \left(\frac{1}{n} \sum_{i=1}^{n} R_i \right)^{-1} \tag{10}$$

where the sum is over the training runs, and R_i is the inverse training time for successful runs and zero otherwise. This measure of training time is more appropriate than taking a simple average, since it includes the effect of network configurations that do not converge to a solution but is not dominated by such contributions. It should be kept in mind, however, that this definition favors those network configurations that converge to a solution more rapidly, and it therefore gives lower values than a simple average would in situations where *all* the network configurations converge to a solution before the cutoff time.

The same criteria were used to measure the success rate and speed of the network reduction algorithm. The results for the speed of the algorithm

parameters				back-prop		reduction		N_R	N_R (as %)			
μ_w	β_{lower}	$f_{\min}$	f_{converge}	τ_{bp}	%	τ_{nr}	%	av.	2	3	4	5+
0.01	−1.0	0.1	0.005	83	100	152	100	2.25	80.5	14.8	4.0	0.7
0.005	"	"	"	50	100	107	100	3.05	39.6	28.6	22.3	9.5
0.02	"	"	"	198	100	275	100	2.07	93.2	6.2	0.6	0.0
0.05	"	"	"	236	98.8	307	98.3	2.14	88.8	8.9	1.9	0.4
0.01	−0.5	"	"	81	100	126	100	2.52	62.6	26.0	8.9	2.5
"	−2.0	"	"	84	100	164	100	2.15	86.9	11.1	1.9	0.1
"	−5.0	"	"	86	98.0	171	97.8	2.12	89.4	9.2	1.4	0.0
"	−1.0	0.05	"	83	100	150	100	2.29	78.4	15.8	4.5	1.3
"	"	0.2	"	83	100	153	100	2.23	81.0	15.5	3.3	0.2
"	"	0.3	"	83	100	155	99.8	2.23	79.4	18.3	2.3	0.0
"	"	0.1	0.002	83	100	220	100	2.13	88.8	9.4	1.5	0.3
"	"	"	0.01	83	100	105	100	2.54	62.9	24.5	8.9	3.7

Table 1: Results for the $N_I = 2$ parity problem (XOR) with $\epsilon = 1.0$, $\alpha = 0.94$. The network starts with 8 units in the hidden layer and the method reduces the net to N_R hidden units.

given here represent the number of steps taken to find a *complete and stable* reduced network configuration; that is, one in which *all* the weights had ceased to change significantly. This criterion gives an *upper* bound on the time required by the algorithm, since most of the elimination of weights and hidden units actually takes place very soon after the learning solution is found. We consider the reduction algorithm to have converged to a solution when the change in *all* weights and biases at any one step is no more than the fraction $f_{\text{converge}} = 0.005$ of the largest weight in the network. The removal of weights and hidden units begins *after* the first network solution is found, and weights which are less than a proportion $f_{\min}$ of the largest weight $w_{\max}$ in that particular layer of the network are removed explicitly. We typically used a value of $f_{\min} = 0.1$; larger values tend to give smaller resultant network solutions, but the algorithm takes longer because more *relearning* must take place after each weight is removed—and larger values of $f_{\min}$ represent, on average, a greater disturbance of the network.

Parity

The $N_I = 2$ parity problem (XOR problem) is one of the simplest problems upon which to test new algorithms. We present results for initial network configurations with 8 units in the hidden layer, which are representative of the behaviour of the algorithm on oversized networks. These results are summarized in Table 1 and represent an average over 1000 random initial network configurations. The percentages for the success rates of back-propagation and the network reduction algorithm are taken with respect to the total number of initial network configurations. The learning rate was chosen to be $\epsilon = 1.0$ and the momentum $\alpha = 0.94$, which represent near-optimal values for batch training using the back-propagation algorithm without any pruning. A cutoff of 1000 presentations of the patterns was used throughout. N_R is the number of hidden units in the reduced net and the minimum possible is two.

The results show a dramatic reduction in the size of the network—the number of hidden units is reduced in the overwhelming majority of cases to 2, and the distribution falls off rapidly towards larger numbers of hidden units. Furthermore, this network reduction is observed for a wide range of parameters—thus eliminating any need for a careful tuning of the parameter values. Although the initial oversized networks could easily allow the "grandmother cell" solution ($N_R = 4$), it is interesting to note that it is only chosen in a small fraction of cases.

The effect of varying each of the parameters can easily be seen. Larger values of the coefficient μ_w of the network reduction contribution give smaller resultant networks, which reflects the influence of this term throughout learning on the extent of the weight space that is explored. However, larger values of the coefficient μ_w also result in longer learning times. Trials were also performed in which the network reduction term was applied only after the network had found a first solution, and the resulting networks were considerably larger than those obtained above. Large negative values of β_{lower} also tend to give smaller network configurations—for much the same reasons. The parameter $f_{\min}$ determines which weights are set explicitly to zero and, as we would expect, larger values give smaller resultant networks—although values that are too large (i.e., larger than 0.3) produce a large disturbance of the network and, sometimes, loss of learning ability (longer solution time and lower success). Likewise, the effect of f_{converge} is straightforward; larger values give faster solution times but also larger resultant networks.

The other network reduction functions given in (9) were also investigated and an overview of the results is given in Table 2. In order to give a comparison of the various functions, all the tests were performed using a learning-rate of $\epsilon = 1.0$, momentum $\alpha = 0.94$, $\beta_{\text{lower}} = -1.0$, $f_{\min} = 0.1$, and $f_{\text{converge}} = 0.005$. The effect of varying these parameters is much the same as we have seen in Table 1, with the same sorts of tradeoffs: smaller resultant networks requiring longer convergence times. These functions give the same qualitative picture as was found previously, but the resultant networks are somewhat larger than those obtained using the simplest network reduction function $E_{\text{nr}} = |w_{ij}|$, and, in general, substantially more iterations were required to find solutions.

The last section of Table 2, labeled "local," denotes the local version of the algorithm, as outlined at the end of Section 2. The resultant networks were somewhat larger and the solution and implementation times were longer. The results obtained using the E_{nr} of (8) were substantially better than all other methods, and it is this function that is used in all subsequent results that we present.

The parity problem with 4 input units and an initial network configuration with 8 hidden units was also investigated. This problem is known to have solutions with 4 hidden units. Back-propagation in the absence of any network pruning is near-optimal for values of the learning rate $\epsilon = 0.8$ and momentum $\alpha = 0.86$, giving an average batch training time of 116 presentations of all training patterns and a success rate of 98.3% (with a cutoff of

$E_{\rm nr}$	μ_w	back-prop $\tau_{\rm bp}$	back-prop %	reduction $\tau_{\rm nr}$	reduction %	N_R av.	N_R (as %) 2	3	4	5+
$\lvert w_{ij}\rvert$	0.01	83	100	152	100	2.25	80.5	14.8	4.0	0.7
w_{ij}^2	0.005	72	100	111	100	4.20	11.0	20.8	27.9	40.3
	0.01	127	99.8	191	99.5	3.01	46.4	26.0	15.0	12.5
	0.02	335	99.8	416	97.4	2.96	47.2	29.1	12.5	11.2
$w_{ij}^2/(1+w_{ij}^2)$	0.02	43	100	70	100	4.38	3.5	18.8	34.3	43.4
	0.05	63	100	95	100	2.82	41.3	39.9	14.6	4.2
	0.1	141	99.8	186	99.8	2.32	73.2	22.0	4.2	0.6
	0.2	237	99.6	279	99.6	2.28	76.3	20.3	2.6	0.8
$1-e^{-w_{ij}^2}$	0.02	44	100	71	100	4.08	5.9	25.5	34.9	33.7
	0.05	64	99.9	94	99.9	2.88	36.1	42.9	17.7	3.4
	0.1	112	99.9	150	99.8	2.41	67.0	26.3	5.8	0.9
	0.2	177	99.9	219	99.9	2.34	70.8	24.8	3.9	0.5
$1-e^{-\lvert w_{ij}\rvert}$	0.02	50	100	78	100	3.47	17.7	36.0	31.1	15.2
	0.05	89	100	125	99.4	2.49	58.1	35.3	6.2	0.4
	0.1	134	99.5	169	99.5	2.35	67.2	30.9	1.9	0.0
	0.2	103	100	138	98.8	2.68	41.5	50.6	6.8	1.1
$\log(1+w_{ij}^2)$	0.02	59	100	89	100	2.95	33.7	43.9	16.7	5.7
	0.05	125	99.5	164	99.4	2.30	72.9	24.6	2.4	0.1
	0.1	160	99.5	197	99.4	2.31	69.8	29.2	0.9	0.1
	0.2	123	99.8	195	90.1	2.49	57.2	37.9	4.0	1.0
local	0.01	45	100	121	100	3.39	16.2	44.4	26.4	13.0
	0.02	56	100	138	99.5	2.86	37.1	44.7	14.5	3.7
	0.03	64	100	165	94.3	2.63	47.6	42.8	8.4	1.2
	0.05	71	100	225	90.9	2.54	49.3	47.3	3.2	0.2

Table 2: Results for the $N_I = 2$ parity problem (XOR) with various network-reduction cost functions. Throughout, $\epsilon = 1.0$, $\alpha = 0.94$, $\beta_{\rm lower} = -1.0$, $f_{\rm min} = 0.1$, and $f_{\rm converge} = 0.005$. The network begins with 8 units in the hidden layer and the method reduces the net to N_R hidden units.

1000 presentations of the learning patterns). These values of the learning rate and momentum are the values chosen for our testing of the network reduction algorithm, although there is a broad range of parameter values that give very nearly the same training performance. The results presented in Table 3 were obtained by averaging over 1000 initial network configurations with a cutoff of 5000 presentations of all the learning patterns.

Network pruning was implemented using (8). This was found to give a substantial reduction in the average size of the resultant networks, but we observed a tendency for the eliminated weights to be distributed over *all* the hidden units. Although this gives a sizable reduction in the number of hidden units, there is still some remaining redundancy. In order to specifically reduce the number of hidden units we introduced a further *hidden unit suppression* factor, whereby hidden units attached to small weights are more highly supressed than those with larger weights. This can be accomplished by associating a factor γ_h with each hidden unit. This factor is incorporated into the change of the incoming weight to that hidden unit as follows (see equation (3)):

$$(\vec{\Omega}_{\rm nr}^1)_{ki} = -\frac{1}{\gamma_k}\frac{\partial E_{\rm nr}}{\partial \vec{w_{ki}}} \tag{11}$$

suppress units	parameters				back-prop		reduction		N_R average
	μ_w	β_{lower}	$f_{\min}$	f_{converge}	τ_{bp}	%	τ_{nr}	%	
no	0.002	-1.0	0.1	0.005	134	98.9	140	98.8	6.56
	0.003	-1.0	0.1	0.005	157	94.7	164	94.6	6.03
yes	0.002	-1.0	0.1	0.005	176	96.4	197	96.3	5.13
	0.003	"	"	"	238	94.4	264	94.1	4.81
	0.005	"	"	"	365	89.7	399	89.3	4.81
	0.003	-0.5	"	"	227	95.5	248	95.3	4.92
	"	-5.0	"	"	240	94.2	269	93.9	4.78
	"	-1.0	0.05	"	238	94.4	260	94.1	4.89
	"	"	0.2	"	238	94.4	284	89.7	4.72
	"	"	0.3	"	238	94.4	371	64.2	4.67
	"	"	0.1	0.002	238	94.4	316	93.8	4.70
	"	"	"	0.01	238	94.4	255	94.2	4.90

Table 3: Results for the $N_I = 4$ parity problem. Throughout, $\epsilon = 0.8$ and $\alpha = 0.86$. The network begins with 8 units in the hidden layer and the method reduces the net to N_R hidden units.

where

$$\gamma_k = \frac{\sum_i |w_{ki}|}{\max_{k'}(\sum_i |w_{k'i}|)} \tag{12}$$

and $k \in$ hidden layer, $i \in$ input layer of units. Thus, the network reduction term will be more significant for weights attached to hidden units whose total incoming weights are smaller. Consequently, such hidden units have a greater tendency to be eliminated from the network. A variety of more complicated expressions for γ_h were also tested, but this particular straightforward expression (11) proved to be the most successful, both for the parity and the symmetry problems.

In Table 3, the first two lines are without any unit suppression term and the remaining results are with the γ_h term as given in equation (11). The substantial reduction that the γ term brings about can be clearly seen. The effect of the other parameters is qualitatively the same as that observed in the case of the XOR problem.

An interesting addendum to our investigations of the parity-4 function is the observation of solutions with three hidden units! Although it is easy to convince oneself that no such solution is possible with a step-function, it is nonetheless entirely possible for such solutions with the sigmoid function to give an arbitrarily small error. These solutions can arise because of very fine tuning of the weights and biases with the sigmoid function. One such solution, in which the difference between the target and actual network output for each pattern was less than 2×10^{-4}, is given by the following network configuration.

On the output layer: $\theta_o = 16.8501,\ w_{o,1} = -26.9472,$
$w_{o,2} = -80.4798,\ w_{o,3} = 48.1751$

Unit 1 in hidden layer: $\theta_1 = -25.9776,\ w_{1,i} = 7.09304$
$\forall i \in$ input layer

suppress nodes	parameters				back-prop		reduction		N_R average
	μ_w	β_{lower}	$f_{\min}$	f_{converge}	τ_{bp}	%	τ_{nr}	%	
no	0.005	-1.0	0.1	0.005	46	97.7	76	97.6	5.77
	0.01	−1.0	0.1	0.005	60	97.1	104	97.1	4.17
	0.02	−1.0	0.1	0.005	131	75.5	162	95.2	3.22
yes	0.002	−1.0	0.1	0.005	43	98.0	87	98.0	4.69
	0.005	"	"	"	59	97.9	131	97.5	3.31
	0.01	"	"	"	100	97.2	149	96.4	2.94
	0.02	"	"	"	165	95.7	203	95.7	2.86
	0.03	"	"	"	316	79.5	375	78.5	2.90
	0.01	−5.0	"	"	103	96.9	158	96.0	2.91
	"	−1.0	0.2	"	238	97.2	150	93.6	2.93

Table 4: Results for the $N_I = 4$ symmetry problem. Throughout, $\epsilon = 2.0$ and $\alpha = 0.85$. The network begins with 8 units in the hidden layer and the method reduces the net to N_R hidden units.

Unit 2 in hidden layer: $\theta_2 = 2.36091,\ w_{2,i} = -1.54448$
$\forall i \in$ input layer

Unit 3 in hidden layer: $\theta_3 = 18.6657,\ w_{3,i} = -12.5477$
$\forall i \in$ input layer

Clearly, there is a whole class of such solutions, although they appeared only rarely.

Symmetry

The symmetry problem with $N_I = 4$ input units, which has a minimal solution of $N_H = 2$ hidden units [1], was also used to test the network reduction algorithm and the results are summarized in Table 4. Again, we started with 8 hidden units and network parameters were determined by averaging over 1000 randomly chosen initial network configurations with a cutoff of 1000 presentations of the 16 learning patterns: $\epsilon = 2.0$ and $\alpha = 0.85$. These parameter values give a training time of 39 complete pattern presentations with a success rate of 97.9%.

The results for the network pruning algorithm are presented in Table 5. The algorithm with the term for suppression of hidden units (12) delivered networks with very nearly the minimal number of hidden units required for the problem.

4. Discussion and conclusions

We have presented and systematically investigated a number of refinements to pruning algorithms. The algorithm for introducing the network reduction contributions ensures that it is not in conflict with the network training algorithm, by making changes to the weights only in directions in weight space that are consistent with learning. This essentially eliminates the problem of spurious minima (due to conflict of the E_{err} and E_{nr} terms) that plagues

pruning algorithms. However, learning may take somewhat longer than for unpruned networks, because the network can move into a region of weight space where the derivative of the error function E_{err} is small.

We have introduced a means not only of pruning weights, but more specifically of pruning weights in such a way that the number of hidden units is significantly reduced. A number of different network reduction functions E_{nr} have also been systematically compared. The algorithms were tested on parity and symmetry problems and they produced considerable reduction in the resultant network size over a large range of parameters.

Acknowledgments

Partial support from the *Sonderforschungsbereich 262* and the Schwerpunkt Schi 257/2-2 is gratefully acknowledged.

References

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation," pp. 318–362 in *Parallel Distributed Processing*, Vol. 1, edited by D. E. Rumelhart and J. L. McClelland (Cambridge, MIT Press, 1985).

[2] S. J. Hanson and L. Y. Pratt, "Comparing Biases for Minimal Network Construction with Back-Propagation," pp. 177–185 in *Advances in Neural Information Processing Systems 1 (NIPS 88)*, edited by D. S. Touretzky (San Mateo, CA, Morgan Kaufmann, 1989).

[3] Y. Chauvin, "A Back-Propagation Algorithm with Optimal Use of Hidden Units," in *Advances in Neural Information Processing Systems 1 (NIPS 88)*, edited by D. S. Touretzky (San Mateo, CA, Morgan Kaufman, 1989).

[4] Y. Chauvin, "Generalization Performance of Overtrained Back-Propagation Networks," pp. 46–55 in *Neural Networks, Proceedings of the EURASIP Workshop, Portugal 1990*, edited by L. B. Almeida and C. J. Wellekens (New York, Springer, 1990).

[5] A. S. Weigend, D. E. Rumelhart, and B. A. Huberman, "Back-Propagation, Weight-Elimination and Time Series Prediction," pp. 105–117 in *Proceedings of the 1990 Connectionist Models Summer School*, edited by D. S. Touretzky, J. L. Elman, T. J. Sejnowski, and G. E. Hinton (San Mateo, CA, Morgan Kaufmann, 1990).

[6] E. D. Karnin, "A Simple Procedure for Pruning Back-Propagation Trained Neural Networks," *IEEE Transactions on Neural Networks*, **1** (1990) 239–242.

[7] M. C. Mozer and P. Smolensky, "Skeletonization: A Technique for Trimming the Fat from a Network via Relevance Assessment," in *Advances in Neural Information Processing Systems 1 (NIPS 88)*, edited by D. S. Touretzky (San Mateo, CA, Morgan Kaufman, 1989).

[8] A. N. Burkitt, "Optimization of the Architecture of Feed-Forward Neural Networks with Hidden Layers by Unit Elimination," *Complex Systems*, **5** (1991) 371–380.

[9] J. Denker, D. Schwartz, B. Wittner, S. Solla, J. Hopfield, R. Howard, and L. Jackel, "Large Automatic Learning, Rule Extraction, and Generalization," *Complex Systems*, **1** (1987) 877–922.

[10] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth, "Occam's Razor," *Information Processing Letters*, **24** (1987) 377–380.

[11] E. B. Baum and D. Haussler, "What Size Net Gives Valid Generalization?" *Neural Computation*, **1** (1989) 151–160.

[12] G. Cybenko, "Complexity Theory of Neural Networks and Classification Problems," pp. 26–44 in *Neural Networks, Proceedings of the EURASIP Workshop, Portugal 1990*, edited by L. B. Almeida and C. J. Wellekens (New York, Springer, 1990).

[13] T. P. Vogl, J. K. Mangis, A. K. Rigler, W. T. Zink, and D. L. Alkon, "Accelerating the Convergence of the Backpropagation Method," *Biological Cybernetics*, **59** (1988) 257–263.

[14] R. Battiti, "Accelerated Backpropagation Learning: Two Optimization Methods," *Complex Systems*, **3** (1989) 331–342.

[15] A. H. Kramer and A. Sangiovanni-Vincentelli, "Efficient Parallel Learning Algorithms for Neural Networks," in *Advances in Neural Information Processing Systems 1 (NIPS 88)*, edited by D. S. Touretzky (San Mateo, CA, Morgan Kaufman, 1989).

[16] F. M. Silva and L. B. Almeida, "Acceleration Techniques for the Backpropagation Algorithm," pp. 110–119 in *Neural Networks. Proceedings of the EURASIP Workshop, Portugal 1990*, edited by L. B. Almeida and C. J. Wellekens (New York, Springer, 1990).

[17] P. J. Gawthorp and D. Sbarbaro, "Stochastic Approximation and Multilayer Perceptrons: The Gain Backpropagation Algorithm," *Complex Systems*, **4** (1990) 51–74.

[18] G. Tesauro and B. Janssens, "Scaling Relationships in Back-propagation Learning," *Complex Systems*, **2** (1988) 39–44.

[19] P. J. G. Lisboa and S. J. Perantonis, "Complete Solution of the Local Minima in the XOR Problem," *Network*, **2** (1991) 119–124.

[20] M. Golea and M. Marchand, "A Growth Algorithm for Neural Network Decision Trees," *Europhysics Letters*, **12** (1990) 205–210.

[21] R. P. Brent, "Fast Training Algorithms for Multilayer Neural Nets," *IEEE Transactions on Neural Networks*, **2** (1991) 346–354.

[22] M. Marchand, M. Golea, and P. Rujan, "A Convergence Theorem for Sequential Learning in Two-Layer Perceptrons," *Europhysics Letters*, **11** (1990) 487–492.

[23] P. Rujan and M. Marchand, "Learning by Minimizing Resources in Neural Networks," *Complex Systems*, **3** (1989) 229–241.

[24] E. B. Baum, "Neural Net Algorithms that Learn in Polynomial Time from Examples and Queries," *IEEE Transactions on Neural Networks*, **2** (1991) 5–19.