Product Unit Learning*

Laurens R. Leerink[a], C. Lee Giles[b,c], Bill G. Horne[b], Marwan A. Jabri[a]

[a]SEDAL, Department of Electrical Engineering
The University of Sydney, Sydney, NSW 2006, Australia
[b]NEC Research Institute, 4 Independence Way, Princeton, NJ 08540, USA
[c] UMIACS, University of Maryland, College Park, MD 20742, USA

Abstract

Product units provide a method of automatically learning the higher-order input combinations required for the efficient synthesis of Boolean logic functions by neural networks. Product units also have a higher information capacity than sigmoidal networks. However, this activation function has not received much attention in the literature. A possible reason for this is that one encounters some problems when using standard backpropagation to train networks containing these units. This report examines these problems, and evaluates the performance of three training algorithms on networks of this type. Empirical results indicate that the error surface of networks containing product units have more local minima than corresponding networks with summation units. For this reason, a combination of local and global training algorithms were found to provide the most reliable convergence.

We then investigate how 'hints' can be added to the training algorithm. By extracting a common frequency from the input weights, and training this frequency separately, we show that convergence can be accelerated.

A constructive algorithm is then introduced which adds product units to a network as required by the problem. Simulations show that for the same problems this method creates a network with significantly less neurons than those constructed by the tiling and upstart algorithms.

In order to compare their performance with other transfer functions, product units were implemented as candidate units in the Cascade Correlation (CC) [13] system. Using these candidate units resulted in smaller networks which trained faster than when the any of the standard (three sigmoidal types and one Gaussian) transfer functions were used. This superiority was confirmed when a pool of candidate units of four different nonlinear activation functions were used, which have to compete for addition to the network. Extensive simulations showed that for the problem of implementing random Boolean logic functions, product units are always chosen above any of the other transfer functions.

1 Introduction

It is well-known that supplementing the inputs to a neural network with higher-order combinations of the inputs both increases the capacity of the network [9] and the the ability to learn geometrically invariant properties [19]. Such a network consists of a single layer of higher-order units [19] which compute the function

$$\theta\left(w_0 + \sum_i w_i x_i + \sum_{i,j} w_{ij} x_i x_j + \sum_{i,j,k} w_{ijk} x_i x_j x_k + \dots\right), \tag{1}$$

where θ is the activation or transfer function.

*Technical Report CS-TR-3503 and UMIACS-TR-95-80, University of Maryland, College Park, MD 20742 (1995)

However, there is a combinatorial explosion of higher order terms as the number of inputs to the network increases. More specifically, the number of weights required to implement a K-th order neuron with N inputs are [32]

$$\sum_{i=0}^{K} \begin{pmatrix} N+i-1 \\ i \end{pmatrix} = \begin{pmatrix} N+K \\ K \end{pmatrix}. \tag{2}$$

Another approach to this problem were 'sigma-pi' networks [35, 30], in which each hidden layer unit calculates a certain product (or conjunct) of the inputs. In this way a polynomial function of the inputs are presented as inputs to the transfer function of the output layer, i.e. the value of the output unit j is

$$\theta\left(\sum_{i \in conjunct} w_{ij} \prod x_{i1} x_{i2} \ldots x_{iN}\right), \tag{3}$$

where i indexes the conjuncts that are used in unit j and assuming there are N values in the conjunct. In this architecture a single higher order term is constructed by each neuron in the hidden layer. This architecture presents another method of constructing higher-order networks, but the problem of the combinatorial explosion of the number of weights still remains if the conjuncts are not hand-coded.

However, Redding et al [33] found that in order to implement a certain logical function, in most cases only a few of these higher order terms are required.

The 'pi-sigma' networks (PSNs) of Ghosh & Shin [18] attempt to make use of this fact. A PSN with N summation units and one 'pi' output neuron provides a K-th order approximation of a continuous function. However, the product units introduced by Durbin & Rumelhart [11] have the advantage that, given an appropriate training algorithm, the units can automatically learn the higher order terms that are required to implement a specific logical function.

In these networks the hidden layer units compute the weighted product (instead of the weighted sum) of the inputs. Each term being multiplied together is a input raised to the power of the variable weight. That is, every unit computes

$$\prod_{i=1}^{N} x_i^{w_i} \qquad \text{instead of} \qquad \sum_{i=1}^{N} x_i w_i. \tag{4}$$

An additional advantage of product units is the increased information capacity of these units compared to standard summation networks. Durbin & Rumelhart [11] determined empirically that the information capacity of these units (as measured by their capacity for learning random Boolean patterns) is approximately $3N$, compared to $2N$ for a single threshold logic function [9]. As before, N is the number of inputs to the network.

The larger capacity is means that the same functions can be implemented by networks containing less units. In many application areas the size of the neural network required to implement the a specific task determines whether a neural network approach is feasible. One example is real-time applications e.g. speech recognition where both the computational load and the bandwidth of the input data is high. Another area is the VLSI implementation of neural networks. In the latter case the size of the network is limited by the amount of space available on the chip, as well as current usage if the network is to be used in a low power application. In this case the number of weights in the network is more important than the number of neurons, as weights require a much larger surface area for implementation [14]. In this case a smaller network with a slightly more complex transfer function is preferable to a larger network with a standard (sigmoidal) transfer function.

When product units are used to process Boolean inputs, best performance is obtained [11] by using values of $+1$ and -1. With these inputs, a product unit computes the following function:

$$\begin{aligned} y &= \prod_{i=1}^{n} x_i^{w_i} && (5) \\ &= e^{\sum_{i=1}^{n} w_i \ln x_i} && (6) \end{aligned}$$

$$
\begin{aligned}
&= e^{\sum_{i|x_i=-1}^{n} w_i \ln x_i} \qquad \text{since} \quad \ln 1 = 0 && (7)\\
&= e^{\sum_{i|x_i=-1}^{n} w_i i\pi} \qquad \text{since} \quad \ln -1 = i\pi && (8)\\
&= \cos \pi \sum_{i|x_i=-1} w_i + i \sin \pi \sum_{i|x_i=-1} w_i && (9)
\end{aligned}
$$

If the imaginary component is ignored, the activation function is equivalent to a cosine summation function with all -1 inputs mapped to 1 and the $+1$ inputs mapped to 0. We can thus use previous results obtained from the study of cosine/sine activation functions to further our understanding of product units. In the remainder of this report the terms *product unit(s)* and *cos(ine)* unit will be used interchangeably as all the problems examined have Boolean inputs.

2 Cosine Activation Functions

Networks with cosine hidden unit activations have been used in the past, and have been shown to have universal approximation properties [28, 16]. Gallant & White [16] proved that a single hidden layer feedforward network using a 'cosine squasher' transfer function can implement a Fourier network. This network implements a Fourier approximation to the desired input-output mapping, and can approximate to any desired degree of accuracy any square integrable function on a compact set using a finite number of hidden units. It then follows that product units are networks with universal approximation properties.

Also relevant are the results of research into the *Vapnic-Chervonenkis (VC) dimension* of different activation functions. In [37] the VC dimension is defined as follows. For a positive integer N, a *dichotomy* (S_-, S_+) on a set $S \subseteq R^N$ is a partition $S = S_- \cup S_+$ of S into two disjoint subsets. A function $f : R^N \rightarrow R$ will be said to implement this dichotomy if it holds that

$$f(u) > 0 \text{ for } u \in S_+ \text{ and } f(u) < 0 \text{ for } u \in S_- \tag{10}$$

The set $S \subseteq R^N$ is *shattered* by f if each dichotomy on S can be implemented by some $f \in F$. The VC dimension is then the *largest* integer l so that at least *some* set S of cardinality l in R which can be shattered by some $f \in F$.

In [37] it was remarked that

> If only μ or $\overline{\mu}$ are desired to be infinite, one may take the simpler example $\theta(x) = \sin(x)$. Note that for all l rationally independent real numbers x_i, the vectors of the form $(sin(\gamma_1 x_1), \ldots, sin(\gamma_l x_l))$, with the γ_i's real, form a dense subset of $[-1, 1]^l$, so all dichotomies on $\{x_1, \ldots, x_l\}$ can be implemented with $(1, sin)$-nets.

where $\overline{\mu}$ specified the VC dimension.

This means that one *sin* unit can implement any dichotomy of l rationally independent (or relatively prime) values, and therefore has an infinite VC dimension.

However, this result cannot be directly applied to product units. The reason is that the argument of the *cos* unit is the result of the inner product between the inputs and the weight vector; in the case of product units with $\{0, 1\}$ inputs this becomes the sum of all weights with nonzero inputs multiplied by π i.e. $\cos \pi \sum_{i|x_i=-1} w_i$. This argument cannot be relatively prime for all 2^N possible input combinations as, when used with $\{0, 1\}$ inputs, the dot product consists of the sums of the different weights. This fact, results in a finite VC dimension for product units.

However, rational independence can be enforced by performing a nonlinear scaling operation on the results of the inner product before applying the *cos* function. This was done by Brady [7] who proposed a learning algorithm for a neural network consisting of only a *single sin* unit. It was shown that this network could implement any discrete function $p : R^n \rightarrow \{0, 1\}$. In practice, however, it is found that a very high numerical precision is required to implement the nonlinear mapping. A finite precision will thus limit the number of inputs to the network and the size of the training set.

It is also worth mentioning that the VC dimension is directly related to the generalization ability of the network. The price that has to be paid for having an infinite VC dimension is that the network cannot be guaranteed to generalize [6]. As most applications of neural networks require this ability, an infinite VC dimension is not always desired.

3 Learning with Product Units

3.1 Training Problems

This section examines the problems that are inherent in training neural networks that contain product units. The aim is to present an overview that will enable product units to become a standard part of the neural network toolbox. In the same way that the hyperbolic tangent or the sigmoid activation functions are preferred for certain applications, we show that product units are very suitable for certain tasks. This is especially the case when a network is required for the implementation of logic functions, where we will show that product units are superior to standard activation functions. These advantages come at the cost of some additional training difficulties, but in this section it will be shown that correct weight initialization combined with a simple extension to the backpropagation algorithm can lower the complexity of learning with product units to the same level as that of sigmoidal activation functions.

As the basic mechanism of a product unit is multiplicative instead of additive, one would expect that standard neural network training methods and procedures cannot be directly applied when training these networks.

This is indeed the case. If a neural network simulation environment is available (in our case the Multi Module Neural Simulation Environment MUME [22] was used), the basic functionality of a product unit can be obtained by simply adding the *cos* function $cos(\pi * input)$ (and it's derivative which is required by backpropagation) to the existing list of transfer functions. This assumes that Boolean mappings are being implemented and the appropriate $\{-1, +1\} \rightarrow \{1, 0\}$ mapping has been performed on the input vectors.

If we then attempt to train a network with 6 inputs, 1 'hidden' product unit and a standard summing output unit network on the parity-6 problem shown in [11], it is found that the standard backpropagation algorithm simply does not work. We have found two main reasons for this:

- Random weight initialization.
- The presence of local minima.

3.1.1 Weight Initialization

The first step in the backpropagation procedure (see [20] p. 120) is to initialize all weights to small random values. The main reason for this is to use the dynamic range of the sigmoid function and it's derivative. For large inputs the sigmoid saturates and the derivative is small, leading to small weight adjustments and slow learning. For the same reason the size of the weights are also often scaled in proportion to the fan-in of the neuron, e.g. proportional to $1/\sqrt{\textit{fan-in}}$.

In contrast to this, the dynamic range of a *cos* unit is unlimited. Initializing the weights to small random values results in an input to the unit in the neighborhood of zero. In this region the derivative is small, so apart from choosing small weights centered around $3n\pi/2$ with $n = \pm 1, \pm 2, \ldots$ this is the worst possible choice.

As the argument of the *cos* in $\cos \pi \sum_{i|x_i=-1} w_i$ is already multiplied by π and is symmetric, the weights were initialized randomly in the range $[-2, 2]$. In fact, learning seems insensitive to the size of the weights, as long as they are large enough.

3.1.2 Local Minima

The presence of local minima when training networks with $(co)sine$ units has been reported earlier. In the discussion of their 'Generalized Fourier Networks' in which *sin* hidden layer activation functions were used, Lapedes & Faber [28] commented that

Further generalizations are possible by considering multilayer networks and different expressions for the transfer function. We point out that using sin's often leads to numerical problems, and nonglobal minima, whereas sigmoids seemed to avoid such problems throughout all our extensive simulations.

This comment summarizes our experience of training with product units. For small problems (less than 3 inputs) backpropagation provides satisfactory training. However, when the number of inputs are increased beyond this number, even with the weight initialization in the correct range, training usually ends up in a local minima. Given the representational power of these units, there certainly is merit in finding an appropriate training algorithm.

3.2 Training Algorithms

With this goal in mind, the following training algorithms were evaluated:

- Backpropagation.
- Simulated Annealing.
- Random Search Algorithm.
- Combinations of the above.

Backpropagation (BP) was used as a benchmark and for using in combination with the other algorithms. It is per definition a local search method. The Jacobs delta-bar-delta learning rate adaptation rule [23] was used along with BP to accelerate convergence, with the parameters were set to the following values ($\theta = 0.35, \kappa = 0.05$ and $\phi = 0.90$).

The random search algorithm (RSA) is a global search method (i.e. the whole weight space is explored during training), and does the following:

- For every epoch, for every weight, generate a new random value.
- Replace the weight by this random value and relax the network.
- If the training error decreases, retain this weight. Otherwise restore old weight value.

Simulated annealing (SA) is a standard optimization method, see ([27], [24],[34]) for more information. The operation of SA is similar to RSA, with the difference that with a decreasing probability solutions are accepted which increase the training error.

The combination of algorithms were chosen (BP & SA, BP & RSA) to combine the benefits of global and local search. Used in this manner, BP is used to find the local minima. If the training error at the minima is sufficiently low, training is terminated. Otherwise, the global method initializes the weights to another position in weight space from which local training can continue.

3.3 Simulation Results

Three problems will be examined in this section,

- The parity N problem with a 2 layer network.
- Computing all logical functions of 3 inputs with 1 product unit.
- Computing random logical functions for various sizes of N.

3.3.1 Parity N

The infamous parity problem is (for the product unit at least) an appropriate task. From the architecture and as illustrated by [11], this problem can be solved by one product unit. The question is whether the training algorithms can find a solution. The architecture simulated has N inputs, 1 product unit in the hidden layer and one summation output unit. All simulations were run for a maximum of 1,000 iterations before being terminated. The target values are $\{-1, +1\}$, and the output is taken to be correct if it has the correct sign. The simulation results are shown in Table 3.3.1.

Parity N	BP # Conv	BP Avg Iter	SA # Conv	SA Avg Iter	RSA # Conv	RSA Avg Iter
6	7	34	10	12.6	10	15.2
8	2	700	10	52.8	10	45.4
10	0	-	10	99.9	10	74.1

Table 1: **Learning the parity problem**: The table shows N, the number of inputs for the parity problem, the number of runs out of 10 that have converged for each algorithm, and the average number of training epochs required for convergence.

For the parity problem it is clear that local learning does not provide good convergence. BP was not combined with either SA or RSA, the addition of local learning to these algorithms can clearly only worsen performance. For the parity problem, global search algorithms have the following advantages:

- The search space is bounded (all weights are initialized in $[-2, +2]$) and only weights in this region are generated.
- The dimension of search space is low (maximum of 11 weights for the problems examined).
- The fraction of the weight space which satisfies the parity problem relative to the total bounded weight space is high (all weights only have to be within a certain margin of each other for the output to have the correct sign).

This is a clear indication that BP alone cannot be used as a training algorithm, and that the addition of a global search procedure is essential for training.

3.3.2 All logical functions of 3 inputs

As part of this investigation, we attempted to reproduce the capacity of $3N$ determined empirically by [11] and compare that to the performance of standard summation units. This was done by running two sets of simulations.

Firstly, one product unit was trained to calculate all $(2^2)^N$ logical functions of the N input variables. Unfortunately, this is only practical for $N \in \{2, 3\}$. For $N = 2$ there are only 16 functions, and a product unit has no problem learning all these functions rapidly with all three training algorithms. In comparison a single summation unit can learn 14 (not the XOR & XNOR functions). For N=3, a product unit is able to implement 208 of the 256 functions, while a single summation unit could only implement 104 (these were the maximum values obtained during training). The simulation results are displayed in Table 3.3.2.

The BP-RSA combination requires further explanation. Several BP-(R)SA combinations were evaluated, but best performance was obtained using a fixed number of iterations of BP (in this case 120) along with one initial iteration of RSA. In this manner BP is used to move to the local minima, and if the training error is still above the threshold value the RSA algorithm generates a new set of random weights from which BP can start again.

The effect of this single initial RSA iteration is to reinitialize all weights to new values. The standard RSA algorithm only changes the weights permanently if a decrease in error is found, but in

BP		SA		RSA		BP-RSA	
# Logic	Avg Iter	# Logic	Avg Iter	# Logic	Avg Iter	# Logic	Avg Iter
189.2	20.5	196.1	43.8	167.4	60.2	208	44.3

Table 2: **Learning all logical functions of 3 inputs**: The table shows the average number of logical functions that could be implemented by one product unit and the average number of training epochs required when training converged. Ten simulations were performed for each of the 256 logical functions, each for a maximum of 1,000 iterations.

many cases extensive searches are required to find weights that are better than that of the current local minima. Fastest training was obtained by simply reinitializing the weights if no convergence has occurred after a certain number of epochs (in most cases this was 120).

It is observed from Table 3.3.2 that BP on it's own learns the fastest, but has the worst performance in terms of the number of logical functions implemented. The SA and RSA algorithms converge surprisingly fast, but with the number of iterations limited to 1,000 the global minima's are not always found. It is clear that the BP-RSA combination provides the best performance. The number of training iterations could be reduced further by initiating the RSA iteration when the BP training error does not decrease significantly, instead of running BP for a fixed number of iterations.

The BP-RSA combination is in effect equivalent to the 'local optimization with random restarts' process discussed by [25, 26], where the local search is this case is performed by the BP algorithm. In [26] it was shown that for certain problems where the error surface was 'exceedingly mountainous', multiple random-start local optimization outperformed simulated annealing. We hypothesize that adding product units to a network makes the error surface sufficiently mountainous so that a global search is required.

4 Training Product Units with 'Hints'

Abu-Mostafa [1] showed that using appropriate 'hints' could decrease learning time in neural networks. Hints are information about the system or training data that is known to the observer, but that is embedded in a non-trivial way in the data. It has also been shown [2] that hints may be extracted from the training data, or by using previous experience learned from solving similar problems.

4.1 Product Unit Computation

When the calculation of one product unit is examined, as shown before, the output of a product unit can be approximated by a *cos* unit if the imaginary component is ignored. For clarity, this is repeated below:

$$\begin{aligned} y &= \prod_{i=1}^{n} x_i^{w_i} && (11)\\ &= \cos \pi \sum_{i|x_i=-1} w_i. && (12)\end{aligned}$$

If the $\{-1,+1\}$ inputs are mapped onto $\{1,0\}$, y then becomes

$$\begin{aligned} y &= \cos\left(\pi \sum_{i=1}^{N} x_i w_i\right) \text{ and} && (13)\\ Error &= \Theta\left[\cos\left(\pi\sum_{i=1}^{N} x_i w_i\right)\right] - \xi, && (14)\end{aligned}$$

$$
\begin{aligned}
&\text{where} && (15)\\
\xi &= \text{the } \{-1,+1\} \text{ target values, and} && (16)\\
\Theta &= \text{the Sign() function .} && (17)
\end{aligned}
$$

The first computation that takes place is an inner product, which maps the input vector onto a one-dimensional input space. The inner product is then scaled by π and the *cos* function applied. The output of the *cos* function should have the same sign as the target value ξ.

The problem of training a *cos* unit can also be considered as follows. We have a set of (x, y) coordinates, where x_i is the result of the inner product and y_i is the corresponding target value ξ_i. The expression $\Theta[\cos()]$ can be seen as a square wave of a to be determined frequency and phase so that it is has the same sign as ξ_i for all values of x_i.

Even if the shifting and pre-scaling is not allowed, it is still possible to extract a common frequency and phase component from the cosine unit. The *cos* calculation can be rewritten as:

$$
\begin{aligned}
y &= \cos\left[\pi \sum_{i=0}^{N} x_i w_i\right] && (18)\\
&= \cos\left[\pi \left(\sum_{i=1}^{N} x_i w_i + w_0\right)\right] && (19)\\
&= \cos\left[\pi \left(w_f \sum_{i=1}^{N} x_i w_i + w_0\right)\right] && (20)
\end{aligned}
$$

The bias weight is now used as the phase of the square wave, and another weight w_f has been added which is the frequency. This is not strictly another free parameter, just a common factor taken out of all the weights.

4.2 The Hints

With the product unit computation in the above form, there are two hints that can be added to the learning process:

- The bias weight w_0 is the phase and is $\{0, 1\}$ depending on whether the target value for the zero input vector is $\{+1, -1\}$. This weight is simply initialized to the correct value.
- The w_f weight can be trained separately using backpropagation, the gradient is given by

$$
\frac{\partial y}{\partial w_f} = -\sin\left[\pi \left(w_f \sum_{i=1}^{N} x_i w_i + w_0\right)\right]\left[\pi \sum_{i=1}^{N} x_i w_i\right]. \qquad (21)
$$

Since it is known that linear independence between the results of the dot-product will increase the probability of a correct w_f being found, an additional hint could be implemented by the introduction of a term in the cost function which maximizes this measure.

4.3 Simulation Results

The performance of this method was first evaluated on the problem of computing all 256 logical functions of 3 variables. The results are shown in Table 4.3. The BP-RSA values from Table 3.3.2 are repeated here for comparison.

The main effect of adding hints is to accelerate training, the performance of the network trained by both algorithms is equal to the maximum capacity of the *cos* unit.

In a second experiment, w_0 was fixed according to the first hint, and $\{w_1 \ldots w_N\}$ initialized to random values. We thus have a network in which only one weight is being trained, w_f.

BP-RSA # Logic	BP-RSA Avg Iter	BP-RSA-HINTS # Logic	BP-RSA-HINTS Avg Iter
208	44.3	208	32.9

Table 3: **Learning all logical functions of 3 inputs**: The table shows the average number of logical functions that could be implemented by one product unit and the average number of training epochs required for when training converged. Ten simulations were performed for each of the 256 logical functions, each for a maximum of 1,000 iterations.

The performance of this network is still respectable, an average of 58.6 logical functions were learned in an average of 487 epochs. The performance will certainly increase if the upper bound of 1,000 epochs is lifted.

As there is only one variable being learned, the error surface can be examined to get some insight into why the learning algorithm behaves the way it does. The error surface for the function which computes the XOR of two inputs, and is zero for all other values (targets are $\{-1, -1, -1, -1, -1, +1, +1, -1\}$ for the 8 input vectors $\{0, 0, 0\} \ldots \{1, 1, 1\}$ respectively) is shown in Figure 1. In this case the weights were initialized to $\{1/2, 1/3 \ldots 1/9\}$.

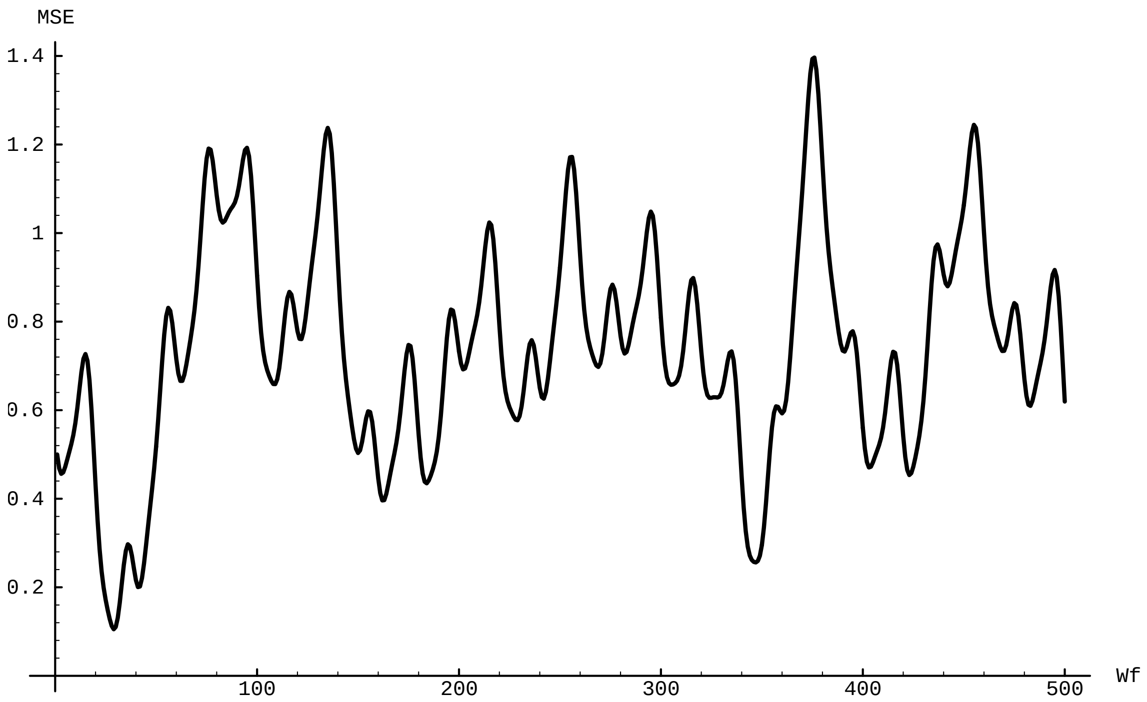

Figure 1: **Error Surface.** This is the error surface when only w_f is trained on the logical XOR of two of three input variables.

From Figure 1 it is clear why standard BP has difficulty in learning this problem, and why the BP-RSA combination performs well. The error surface is locally smooth, and BP would rapidly find the local minima. The RSA step would then position w_f at another (random) value if the training error at the local minima is not sufficiently low.

5 Constructive Learning with Product Units

Selecting the optimal network architecture for a specific application is a nontrivial and time-consuming task, and several algorithms have been proposed to automate this process. One class of network adaptation algorithms start out with a redundant architecture and proceed by pruning away seemingly unimportant weights ([36], [29]). Another class of algorithms start off with a sparse architecture and grows the network to the complexity required by the problem. Several algorithms have been proposed for growing feedforward networks. The 'upstart' algorithm of Frean [15] and the 'cascade-correlation' algorithm of Fahlman [13] are examples of this approach.

In this section we will use the experience gained from the experiments described earlier to devise a simple method for adding product units to a three layer network. The hidden layer will consist of product units, while the output layer contains a single sigmoidal unit.

5.1 The Constructive Algorithm

Several constructive algorithms ([15], [13], [31]) are based on the following process:

- A network is trained on a certain training set.
- The current weights in the network are frozen, and a new neuron (or a set of candidate neurons) is added to the network. Often the new neuron is a fully connected hidden layer.
- The new weights associated with the neuron are now trained on either the whole or a smaller subset of the training set.
- As neurons are added, the performance of the network improves. For some training algorithms proofs are available that specify that, for a given accuracy, the number of neurons to be added is finite.

From our simulations (and that of others, see [28]) it is clear that networks containing *cos* transfer functions suffer from serious local-minima problems, something which sigmoids are reasonably insensitive to. The importance of the RSA (or global search) step when training product units is clear from our experiments, and freezing all the weights (and thereby limiting the global search only to the one new weights) limits the learning process. This is because the addition of a new weight adds one degree of freedom, but limits the new solution to an affine subset of the existing weight space [5]. When a new hidden unit is added, it is possible to rotate the previous solution so that the local minima is reached. However, as shown in [5] this network will not be of minimal size.

For this reason a simple constructive approach was implemented which retains the global (RSA) search for all weights during the whole training process. The method used in our simulations is as follows:

- Train a network using the BP-RSA combination with 'hints' on a network with a specified minimum number of hidden product units.
- If there is no convergence within a specified number of epochs, add a product unit to the network. Reinitialize weights and continue training with the BP-RSA combination.
- This process is repeated until a maximum number of epochs is reached or the network has grown to the maximum specified size.

The method of [8] and [5] was also evaluated, in which neurons with small weights were added to a network according to certain criteria. The method outlined above performed better, possibly because of the global search performed by the RSA step.

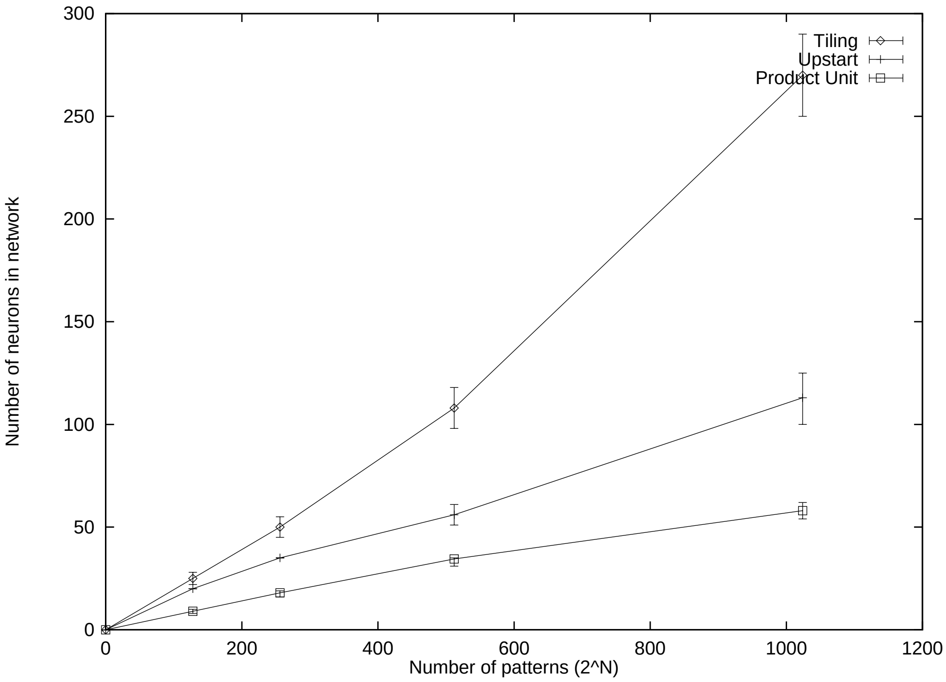

Figure 2: **Learning Random Mappings**: Number of units added to the network when trained on 2^N random mappings. The values are the average of 25 simulations, each on a different training set.

5.2 Simulations

To evaluate the performance of the method with that of the 'upstart' [15] and 'tiling' [31] algorithms, the constructive product network was trained on two problems also described in these papers namely

- The parity problem, for $N = 7 \ldots 10$.
- Random Mappings.

5.2.1 The Parity Problem

In [15] it was reported that the upstart algorithm required N units for all parity N problems, and 1,000 training epochs were sufficient for all values of N except $N = 10$, which required 10,000.

As shown in an earlier section, a product unit is able to perform any parity function with only one product unit. The training iterations required for $N = 6, 8, 10$ were given in Table 3.3.1, for interest the RSA algorithm required an average of 74.1 iterations. In this case our BP-RSA algorithm settings are modified so that more RSA iterations are performed compared to the BP iterations.

5.2.2 Random Mappings

Following [15], in this problem the random mapping problem is defined by assigning each of the 2^N patterns its target $\{-1, +1\}$ with 50% probability. As mentioned in [15], this is a difficult problem, due to the absence of correlations and structure in the input for the network to exploit. As in [15, 31] the average of 25 runs were performed, each on a different training set.

The number of units added by the upstart algorithm was approximately $2^N/9$. The tiling algorithm requires approximately double that amount while the number of units added by the constructive product network are approximately half. The values are plotted in Figure 2. Note that the values for the tiling and upstart algorithms are approximate and were obtained through inspection from a similar graph in [15].

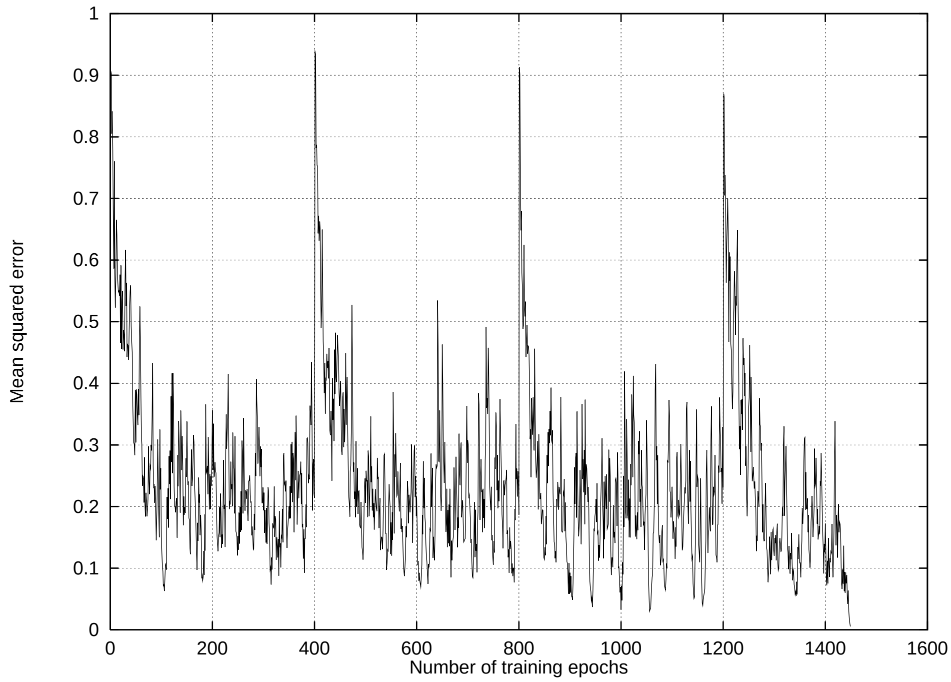

Figure 3: **Training Error During Constructive Learning**: The mean squared error during learning; for this specific problem 3 product units were added at 400 epoch intervals.

5.3 Discussion

As expected, the higher separating capacity of the product unit enables the construction of networks with less neurons than those produced by the tiling and upstart algorithms.

The fact that a simple incremental method works this well is mainly a result of the error surface; the surface is so irregular that even training a network of fixed architecture is best done by reinitializing the weights if convergence does not occur within certain bounds. This again is in accordance with the results of [25, 26] who showed that for certain problems where the error surface was 'exceedingly mountainous', multiple random-start local optimization outperformed more sophisticated methods.

For the training of the random mapping problem, it was found that the parameter settings of the Jacobs [23] delta-bar-delta weight update rule was critical in ensuring rapid convergence. The parameters were set to the following values ($\theta = 0.35, \kappa = 0.05$ and $\phi = 0.90$) for $N = 7, 8, 9$ and $\kappa = 0.003$ for $N = 10$.

The parameter settings of the delta-bar-delta weight update rule influence how the decent in error space is performed. The parameter settings causing the irregular decent shown in Figure 3 not only speeds up learning but was found to be necessary for convergence. We hypothesize that this stochasticity is necessary for rapid training and avoiding local minima.

There is also a tradeoff between obtaining fast convergence and finding a minimum network for the task. In Figure 3 it can be seen that the network could probably have learned the problem with only two additional units if training (and possibly a RSA iteration) had been continued for more than 400 iterations before the addition of the third unit.

Theoretical capacity analysis using methods from statistical mechanics [17, 3, 20, 12] provide some insight into this process. The fraction of weight space that contains a solution depends on:

- The capacity (information theoretic) of the network.
- The complexity of the problem.

This means that if we want to find a minimum size network for a certain problem, given a finite numerical precision there might only be a few solutions in weight space. Given a mountainous error surface, a local search algorithm is ineffective and an exhaustive search is required to find the correct solution. As the network becomes larger, the fraction of weight space that contains solutions to the problem increases, and the learning speed increases correspondingly as these solutions are easier to find. However, the danger of overtraining increases and generalization might suffer.

In the method proposed, the extent of the global search is limited by the number of random restarts that are permitted for a certain architecture. If no solution is found, the learning problem is simplified by adding another hidden unit. The tradeoff between rapid learning and a minimal architecture can thus be controlled through one variable, the number of random restarts.

5.4 Using Cosine Candidate Units in Cascade Correlation

5.4.1 Introduction

The Cascade Correlation (CC) [13] algorithm is a well-known constructive method, and has been applied to both feedforward and recurrent networks. It has been shown [21] that for the neural network implementation of n-input, m-output logic functions (the type of problem examined here) the node-complexity of the fully connected architecture used in CC is $O(\sqrt{\frac{m2^n}{n-\log m}})$. For the one hidden layer units used in SIM the node complexity increases to $O(2^n + m)$. Thus the size of the networks constructed by CC and SIM cannot be directly compared.

However, since CC is a popular algorithm it was thought interesting to compare the performance of our simple method with it. In fact, CC performed substancially better than the upstart and tiling algorithms discussed earlier.

As the resulting networks could not be compared directly, it was decided to evaluate product units in a fully connected structure. The simplest method was to implement these units as candidate units inside CC. This also allows direct comparison with other transfer functions implemented in CC, as well as the construction of hybrid networks consisting of mixtures of candidate units.

During the training process, the CC algorithm constructs a fully connected (lower triangular connection matrix) neural network, with each hidden unit recieving input both from the inputs and all preceding hidden units. The hidden units are added to the network one at a time, and are selected from a pool of 'candidate units', which have to compete for addition to the network. Each of these candidate units are initialized with a different set of random initial weights, and may also be different nonlinear functions.

The public domain version of CC used [38] supports four different candidate types; the asymmetric sigmoid, symmetric sigmoid, variable sigmoid and gaussian units. Facilities exist for either selecting one unit type, or training with a pool of different units allowing the construction of hybrid networks. It was thus relatively simple to add a cosine candidate unit to the system.

5.4.2 The Simulation Results

The random mapping problem was chosen to compare the performance of CC with SIM. It is sufficiently complex, and requires larger networks than most other 'toy problems'. It also allows comparisons with the upstart and tiling algorithms.

The performance of standard CC in terms of number of hidden units required is shown in Figure 4. Four graphs are shown, from top to bottom these are firstly SIM, followed by CC using sigmoidal, Gaussian and finally cosine candidate units.

In a separate experiment the performance of hybrid networks were evaluated on the same random logic problem. When the default settings are used, this implementation of CC constructs a homogeneous network consisting of sigmoidal hidden units. At each step 8 different candidate units compete for addition to the network. After the cosine units had been added to the network, there were 5 different types of candidate units available.

To enable a fair competition between candidate units of different types, the simulations were run with 40 candidate units, 8 of each type. The simulations were re-evaluated on 25 trails for each of

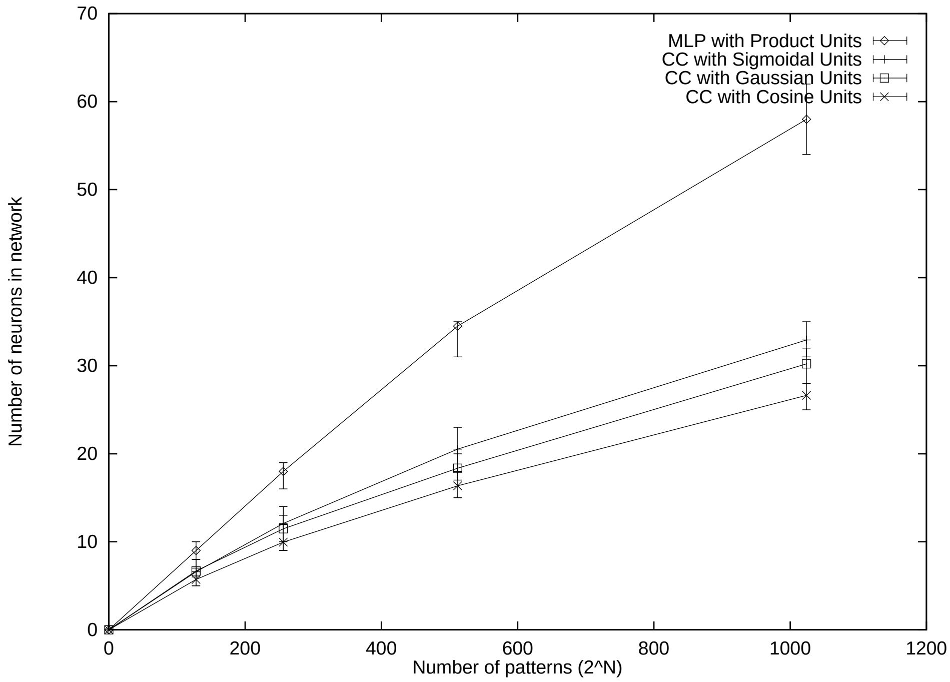

Figure 4: **Learning Random Mappings**: Number of units added to the network when trained on 2^N random mappings. The values are the average of 25 simulations, each on a different training set. The vertical bars indicate the minimum and maximum values.

the random mapping problems (7,8,9 and 10 inputs, a total of 1920 input vectors). In total 1460 hidden units were allocated, and in *all cases* cosine candidate units were chosen above units of the 4 other types during the competitive stage.

During this comparison, all parameters were set to default values, i.e. the weights of the cosine units were random numbers initialized in the range of $[-1, +1]$. As discussed earlier, this puts the cosine units at a slight disadvantage as their optimum range is $[-2, +2]$.

In terms of epochs required for convergence, Table 5.4.2 displays the results when CC was trained using three types of homogeneous candidate units.

N	CC Sigmoid		CC Gauss		CC Cosine	
	$\overline{n}_{epochs}$	σ_{epochs}	$\overline{n}_{epochs}$	σ_{epochs}	$\overline{n}_{epochs}$	σ_{epochs}
7	924.5	104.45	642.6	109.6	493.8	102.2
8	1630.9	164.0	1128.2	112.3	833.8	82.4
9	2738.3	164.1	1831.1	93.3	1481.8	175.9
10	4410.9	164.4	2967.6	98.9	2590.8	167.4

Table 4: **Number of Training Epochs**: The table shows the average and standard deviation of the number of training epochs required for convergence of CC using Sigmoidal, Gaussian and Cosine candidate units.

5.4.3 Discussion

We hypothesize that there are two reasons for the choice of cosine units above any of the other types during the competitive learning phase:

- The higher capacity (in a information capacity sense) of the cosine units, allowing a better correlation with the error signal.
- Although the error surface of the network with cosine units contain more local minima than the error surface of other transfer functions, the surface is locally smooth. This allows fast convergence by the quickprop algorithm.

In [10] it was shown that networks with Gaussian units train faster and require less units than networks with standard sigmoidal units. This is supported by our results as shown in Figure 4. However, for the problem examined, cosine units outperform Gaussian units by approximately the same margin as Gaussian units outperform sigmoidal units.

These results confirm earlier conclusions which suggest that these units are well suited to learning certain types of classification problems, and should be concidered as alternative transfer functions. It should also be noted that these problems where not chosen for their suitability for product units. In fact, if the problems are symmetric the difference in performance is expected to increase.

6 Conclusion

Of the learning algorithms examined, BP combined with the delta-bar-delta weight update rule provides the fastest training, but is prone to nonglobal minima. On the other hand, global search methods perform well on small problems, but are impractical for larger networks. Given a network containing product units, there are some atypical heuristics that can be used: (a) weights leading into the product unit have to be initialized in the range $[-2\pi, 2\pi]$ (b) reinitialization of the weights if convergence is not reached after a certain number of epochs.

The representational power of product units have enabled us to solve standard problems with construction of significantly smaller networks than previously reported, using a very simple constructive method.

When implemented in the well-known cascade correlation architecture, product units resulted in smaller networks which trained faster than the three sigmoidal and one Gaussian unit found in standard CC. In the CC framework a pool of candidate units of different types can be created, allowing competition between the various hidden unit functions. Simulations show that in all cases cosine candidate units were preferred over candidate units of the other four types.

References

[1] Y. Abu-Mostafa, "Learning from hints in neural networks," *Journal of Complexity*, vol. 6, p. 192, 1990.

[2] K. Al-Mashouq and I. Reed, "Including hints in training neural nets," *Neural Computation*, vol. 3, no. 3, pp. 418–427, 1991.

[3] D. Amit, *Modelling Brain Function.* Cambridge: Cambridge University Press, 1989.

[4] J. Anderson and E. Rosenfeld, eds., *Neurocomputing: Foundations of Research.* Cambridge: MIT Press, 1988.

[5] T. Ash, "Dynamic node creation in backpropagation networks," *Connection Science*, vol. 1, no. 4, pp. 365–375, 1989.

[6] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth, "Learnability and the vapnik–chervonenkis dimension," *Journal of the Association for Computing Machinery*, vol. 36, no. 4, pp. 929–965, 1989.

[7] M. Brady, "Guaranteed learning algorithm for network with units having periodic threshold output function," *Neural Computation*, vol. 2, pp. 405–408, 1990.

[8] D. Chen, C. Giles, G. Sun, H. Chen, Y. Lee, and M. Goudreau, "Constructive learning of recurrent neural networks," in *1993 IEEE International Conference on Neural Networks*, vol. III, (Piscataway, NJ), IEEE Press, 1993.

[9] T. Cover, "Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition," *IEEE Transactions on Electronic Computers*, vol. 14, pp. 326–334, 1965.

[10] M. Dawson and D. Schopflocher, "Modifying the generalized delta rule to train networks of nonmonotonic processors for pattern classification," *Connection Science*, vol. 4, pp. 19–31, 1992.

[11] R. Durbin and D. Rumelhart, "Product units: A computationally powerful and biologically plausible extension to backpropagation networks," *Neural Computation*, vol. 1, pp. 133–142, 1989.

[12] A. Engel, H. Kohler, F. Tschepke, and A. Zippelius, "Storage capacity and learning algorithms for two-layer neural networks," *Physical Review A*, vol. 45, pp. 7590–7609, 1992.

[13] S. Fahlman and C. Lebiere, "The cascade-correlation learning architecture," in *Advances in Neural Information Processing Systems* (D. Touretzky, ed.), vol. 2, (San Mateo), pp. 524–532, (Denver 1989), Morgan Kaufmann, 1990.

[14] B. Flower and M. Jabri, "Single and Dual Transistor Synapses for Analogue VLSI Artificial Neural Networks," in *Proceedings of the Fourth Australian Conference on Neural Networks*, pp. 106–109, 1993.

[15] M. Frean, "The upstart algorithm: A method for constructing and training feedforward neural networks," *Neural Computation*, vol. 2, pp. 198–209, 1990.

[16] A. Gallant and H. White, "There exists a neural network that does not make avoidable mistakes," in *IEEE International Conference on Neural Networks*, vol. 1, (New York), pp. 657–664, IEEE, 1988.

[17] E. Gardner, "The space of interactions in neural network models," *Journal of Physics A*, vol. 21, pp. 257–270, 1988.

[18] J. Ghosh and Y. Shin, "Efficient higher-order neural networks for function approximation and classification," *International Journal of Neural Systems*, vol. 3, no. 4, pp. 323–350, 1992.

[19] C. Giles and T. Maxwell, "Learning, invariance, and generalization in high-order neural networks," *Applied Optics*, vol. 26, no. 23, pp. 4972–4978, 1987.

[20] J. Hertz, A. Krogh, and R. Palmer, *Introduction to the Theory of Neural Computation*. Redwood City, CA: Addison–Wesley, 1991.

[21] B. Horne, *Recurrent neural networks: A functional approach*. PhD thesis, University of New Mexico, Dept. of Electrical and Computer Engineering, May 1993.

[22] M. Jabri, E. Tinker, and L. Leerink, "MUME – an environment for multi–net and multi–architectures neural simulation," in *Neural Network Simulation Environments* (J. Skrzypek, ed.), Norwell, MA: Kluwer Academic Publications, 1994.

[23] R. Jacobs, "Increased rates of convergence through learning rate adaptation," *Neural Networks*, vol. 1, pp. 295–307, 1988.

[24] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon, "Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning," *Operations Research*, vol. 37, pp. 865–891, 1989. Parts II and III expected to appear in 1990.

[25] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon, "Optimization by simulated annealing: an experimental evaluation; part ii, graph coloring and number partitioning," *Operations Research*, vol. 39, no. 3, pp. 378–406, 1991.

[26] N. Karmarkar and R. Karp, "The differencing method of set partitioning," Tech. Rep. UCB/CSD 82/113, Computer Science Division, University of California, Berkeley, California, 1982.

[27] S. Kirkpatrick, C. G. Jr., , and M. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, 1983. Reprinted in [4].

[28] A. Lapedes and R. Farber, "Nonlinear signal processing using neural networks: Prediction and system modelling," Tech. Rep. LA–UR–87–2662, Los Alamos National Laboratory, Los Alamos, NM, 1987.

[29] Y. Le Cun, J. Denker, and S. Solla, "Optimal brain damage," in *Advances in Neural Information Processing Systems* (D. Touretzky, ed.), vol. 2, (San Mateo), pp. 598–605, (Denver 1989), Morgan Kaufmann, 1990.

[30] G. L. Maxwell T., G.L Giles and H. Chen, "Nonlinear dynamics of artificial neural systems," in *Neural networks for computing* (J. Denker, ed.), New York: American Institute of Physics, 1986.

[31] M. Mézard and J.-P. Nadal, "Learning in feedforward layered networks: The tiling algorithm," *Journal of Physics A*, vol. 22, pp. 2191–2204, 1989.

[32] M. Minsky and S. Papert, *Perceptrons.* Cambridge: MIT Press, 1969. Partially reprinted in [4].

[33] N. Redding, A. Kowalczyk, and T. Downs, "A constructive higher order network algorithm that is polynomial-time," *Neural Networks*, vol. 6, p. 997, 1993.

[34] F. Romeo, *Simulated Annealing: Theory and Applications to Layout Problems.* PhD thesis, University of California at Berkeley, 1989. Memorandum UCB/ERL–M89/29.

[35] D. Rumelhart, G. Hinton, and J. McClelland, "A general framework for parallel distributed processing," in *Parallel Distributed Processing* (D. Rumelhart and J. McClelland, eds.), vol. 1, ch. 2, pp. 45–76, Cambridge: MIT Press, 1986. Reprinted in [4].

[36] J. Sietsma and R. Dow, "Neural net pruning—why and how," in *IEEE International Conference on Neural Networks*, vol. 1, (New York), pp. 325–333, (San Diego 1988), IEEE, 1988.

[37] E. Sontag, "Feedforward nets for interpolation and classification," nar, Rutgers University, April 1991.

[38] M. White, "A public domain C implemention of the Cascade Correlation algorithm," (Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA), 1993.