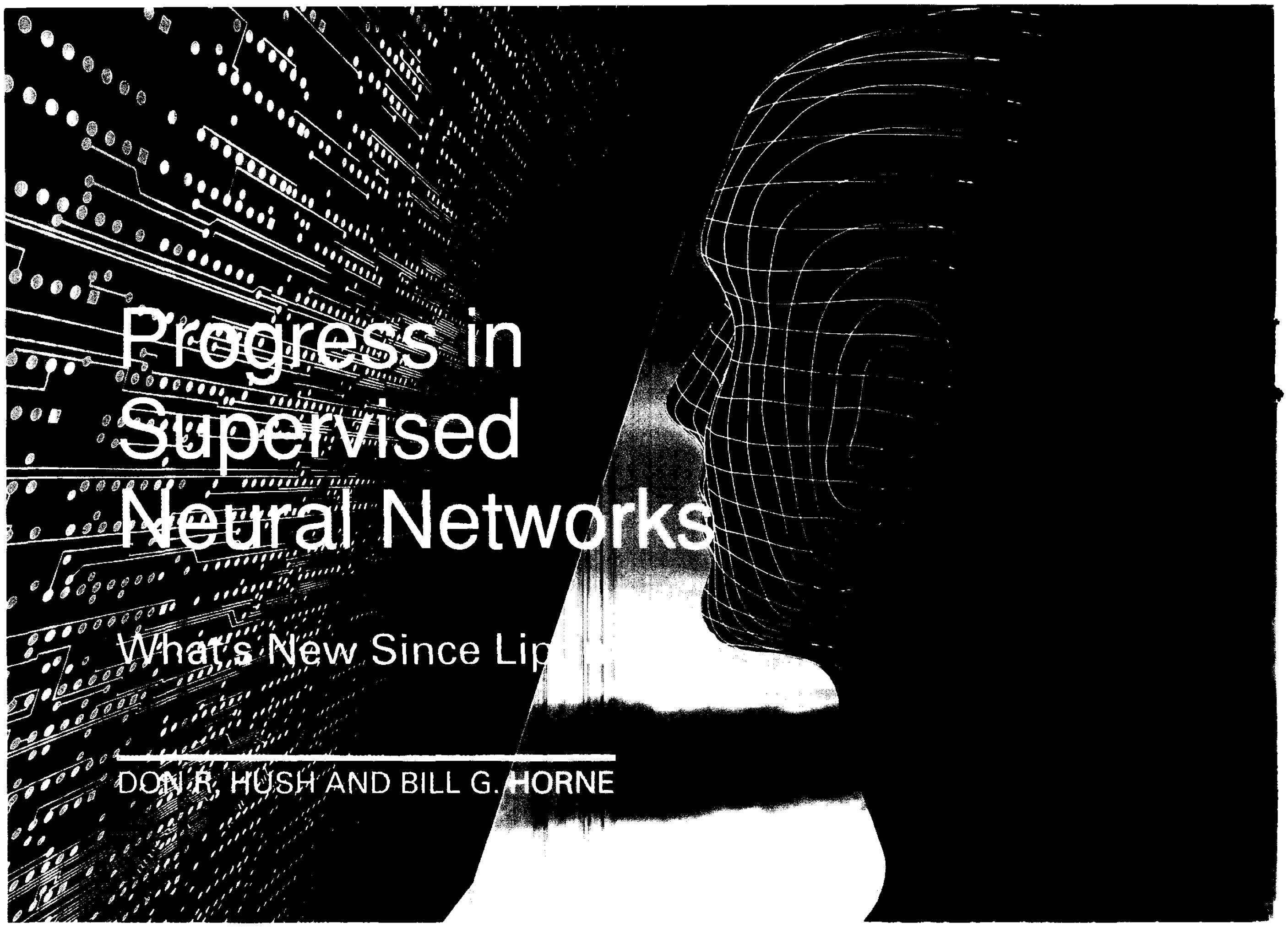


Progress in Supervised Neural Networks

What's New Since Li

DON R. HUSH AND BILL G. HORNE

Uniphoto

Since Richard Lippmann's 1987 tutorial article in ASSP Magazine *An Introduction to Computing with Neural Networks* [81], a number of additional tutorials have appeared, but none have compared in popularity to the original. Even today, Lippmann's article is one of the most widely referenced papers in the neural network literature.

In the original article, Lippmann discussed several different neural network models, the composition of which form the foundation for most of today's neural network research. Two of the newest and most theoretically immature at the time were the Hopfield network and the multilayer perceptron (MLP). There were many unresolved issues concerning these two models, such as finding the best way to *learn* the weights in the Hopfield network; using the Hopfield network to process time-varying inputs; finding the number of layers needed in a multilayer network; determining the intrinsic difficulty of learning in multilayer networks; determining the conditions for which we could expect to achieve good generalization in multilayer networks; finding how to use multilayer networks to estimate probability functions; and the efficacy of multilayer networks in scaling to larger problems.

Significant progress has been made on these and related issues in the past few years. In addition, numerous extensions to the basic models in Lippmann's paper have emerged. Most of these were designed to help overcome some of the inherent limitations of the basic models. For example, extensions to the MLP have made it possible for this network to solve larger and more difficult problems. Extensions to the Hopfield (and other) networks have generated interest in a new class of dynamic network models called *recurrent neural networks* which are capable of performing a wide variety of computational tasks including sequence recognition, trajectory following, nonlinear prediction, and system modeling.

Our goal in this article is to summarize the recent theoretical results concerning the capabilities and limitations of these models, and to discuss some of their extensions. The network models that we discuss are partitioned into two basic categories: static networks and dynamic networks. Static networks, of which the MLP is the most widely used, are characterized by node equations that are memoryless. That is, their output is a function only of the current input, not of past or future inputs or outputs. Another static model that has gained a great deal of notoriety in recent years is the Radial

Basis Function (RBF) network. It is popular and provides an interesting contrast to the MLP.

Dynamic networks, on the other hand, are systems with memory. Their node equations are typically described by differential or difference equations. They can be categorized into three different groups: networks with feedforward dynamics, networks with output feedback, and networks with state feedback. Networks from each of these groups will be discussed, with the greatest emphasis placed on networks with state feedback.

For the most part, the networks discussed in this article are trained using *supervised* learning. This means they are presented with a set of example input-output pairs (x_i, d_i) and trained to implement a mapping that matches the examples as closely as possible. In contrast, for *unsupervised* learning the networks are presented with only the input samples, x_i, and samples are grouped into classes which are self-similar. Networks trained in this fashion are called self-organizing networks, examples of which are the Adaptive Resonance Theory (ART) network and Kohonen's Self Organizing feature maps.

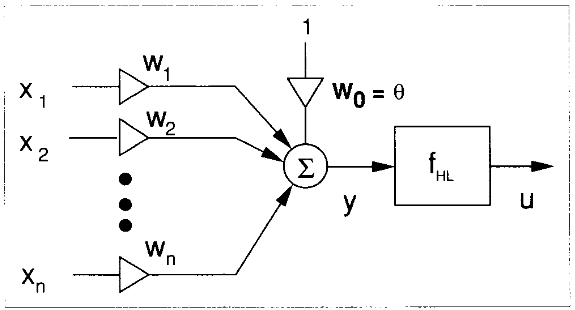
Static Networks

Static networks implement nonlinear transformations of the form $\mathbf{u} = G(\mathbf{x})$. Typically $\mathbf{x} \in \Re^n$, and $\mathbf{u} \in [0,1]^m$ or $\mathbf{u} \in \Re^m$, where n and m are integers that represent the dimensions of x and u, respectively. These networks are useful in a variety of applications.

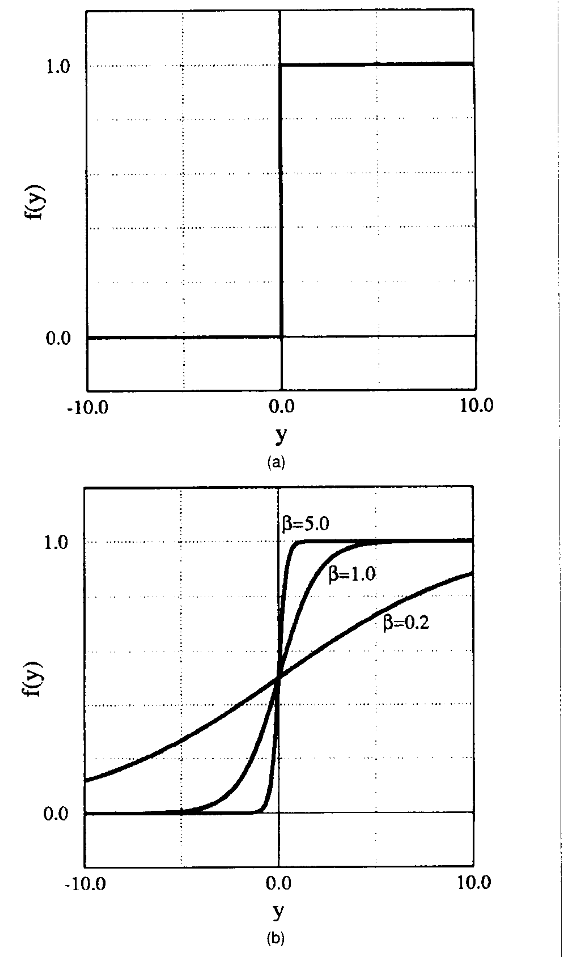
We present examples of their application to logic functions, pattern recognition, and functional approximation. The static networks discussed in the sections that follow include the multilayer perceptron (MLP), its extensions, and the radial basis function (RBF) network. Our discussion will focus on the following characteristics of these models: the model itself, its functional capabilities, learning algorithms for the model, the inherent complexity of learning, and generalization and its relationship to sample-size complexity. We begin our discussion of static networks with the *perceptron*, which is the building block of the MLP network.

The Perceptron

The perceptron (Fig. 1) was conceived by Rosenblatt in 1958 [118]. The input is an n-dimensional vector. The perceptron



1. The Perceptron.



2. Nonlinear activation functions. Hard-limiting (a); sigmoid (b).

forms a weighted sum of the n components of the input vector and adds a *bias value*, θ (alternatively, we could view this as subtracting a threshold value, - θ). The result is then passed through a nonlinearity (Fig. 2). Rosenblatt's original model used the hard-limiting nonlinearity (Fig. 2a), whereby:

$$f_{HL}(y) = \begin{cases} 1 & y > 0 \\ 0 & y \leq 0 \end{cases} \tag{1}$$

When perceptrons are cascaded together in layers, it is more common to use the sigmoid nonlinearity shown in Fig. 2b:

$$f_s(y) = (1 + e^{-\beta y})^{-1} \tag{2}$$

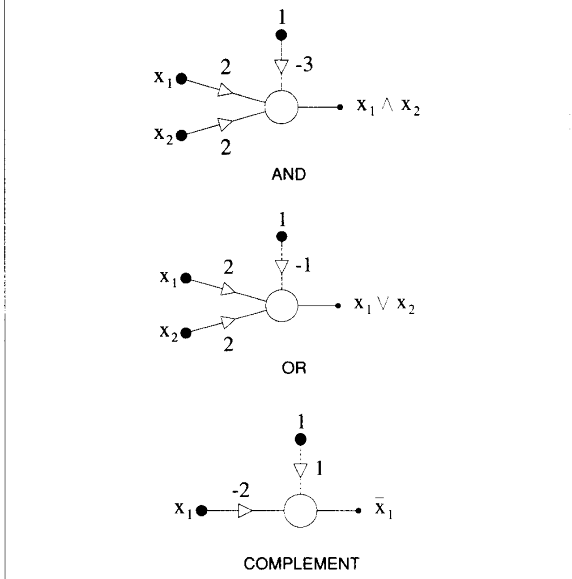
This function is continuous and varies monotonically from 0 to 1 as y varies from $-\infty$ to ∞. The gain of the sigmoid, β, determines the steepness of the transition region. Note that as the gain approaches infinity, the sigmoid approaches a hard-limiting nonlinearity. Often the gain is set equal to 1. One of the advantages of the sigmoid is that it is *differentiable*. This property had a significant impact historically because it made it possible to derive a gradient search learning algorithm for networks with multiple layers.

The sigmoid nonlinearity is popular for other reasons as well. For example, many applications require a continuous-valued output rather than the binary output produced by the hard-limiter. In addition, it is particularly well-suited to pattern recognition problems because it produces an output between 0 and 1 that can often be interpreted as a probability estimate [114]. These topics will be discussed further in subsequent sections. For now we will concentrate on Rosenblatt's original model, which uses a hard-limiter.

For the sake of notation, it will be convenient to view the threshold as a *bias weight*, $w_0 = \theta$, and the input vector as being augmented with an additional dimension which always assumes a value of unity. With this convention we can represent the weighted sum as an inner product between the augmented input vector and the weight vector. Thus the operation performed by the perceptron can be expressed as:

$$\begin{aligned} y &= \boldsymbol{w}^T \boldsymbol{x} \\ u &= f_{HL}(y) \end{aligned} \tag{3}$$

There are two ways of viewing the operation of a perceptron. First, it can be viewed as a discriminant function for two-class pattern recognition problems; that is, as a function which performs a nonlinear transformation from $\mathbf{x} \in \Re^n$ to $u \in \{0,1\}$ according to the class assignment of $\mathbf{x}$. When viewed in this manner, the perceptron effectively partitions the input space into two regions with a linear decision boundary. This is easily verified since points that lie on the boundary between the two classes are described by $y = \mathbf{w}^T\mathbf{x} = 0$, which is the equation of an $(n - 1)$ dimensional hyperplane in $\Re^n$. As a result, the perceptron is suitable as a discriminant function in classification problems which are best solved by a linear dichotomization of the pattern space.



3. Simple logic functions implemented by the perceptron (hard-limiters not shown).

There are many interesting problems however, that require a nonlinear partitioning of the pattern space. These can be solved with the multilayer perceptron network, which cascades two or more layers of perceptrons together, thus making it possible to partition the pattern space with arbitrarily complex decision boundaries.

Second, the perceptron can be viewed as a binary logic unit. It is capable of implementing numerous logic functions, including the three fundamental operations of Boolean algebra: AND, OR and COMPLEMENT (Fig. 3). The perceptron cannot, however, implement *all* possible logic functions. For example, it cannot implement the exclusive-or (XOR) function. With n variables, there are a total of 2^{2^n} logic functions (2^n rows in the truth table yield 2^{2^n} combinations of output functions). The fraction of these that the perceptron can implement are called *threshold logic functions*. Deriving an exact expression for the number of threshold logic functions, NTL(n), has proven an illusive task. However, the following bounds are well-known [96]:

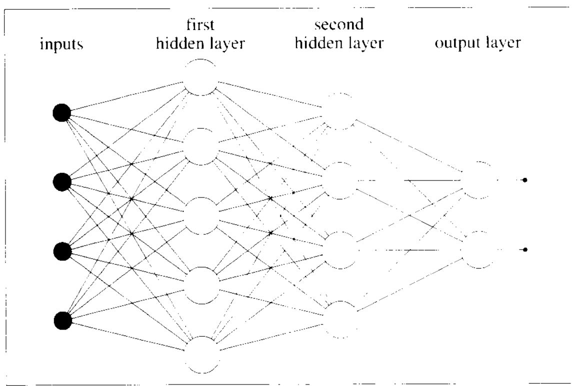
$$2^{n(n-1)/2} < NTL(n) \le 2\sum_{i=0}^{n} \binom{2^n - 1}{i} < 2^{n^2} \tag{4}$$

This is a vanishingly small percentage of the total number of logic functions for large n.

There are numerous learning algorithms for the perceptron. Most of them were developed in the 1960s. They include the perceptron learning algorithm [118], the Least Mean Squares (LMS) learning algorithm [149], and many others [28, 134]. For the most part, the details of these algorithms are beyond the scope of this article. The LMS algorithm, however, is a special case of the *backpropagation* learning algorithm (for multilayer perceptrons), and will be discussed shortly.

The Multilayer Perceptron (MLP)

The capabilities of single perceptrons are limited to linear decision boundaries and simple logic functions. However, by cascading perceptrons in layers we can implement complex decision boundaries and arbitrary Boolean expressions. The individual perceptrons in the network are called *neurons* or *nodes*, and differ from Rosenblatt's perceptron in that a sigmoid nonlinearity is commonly used in place of the hard-limiter. The input vector feeds into each of the first layer perceptrons, the outputs of this layer feed into each of the second layer perceptrons, and so on (Fig. 4). Often the nodes are fully connected between layers, i.e., every node in layer l is connected to every node in layer $l+1$. Sometimes the input vector itself is also referred to as a layer of the network, although we prefer not to do so here.



4. *Architecture of a typical multilayer perceptron.*

Thus we refer to the architecture in Fig. 4 as a 3-layer network. It is also common to specify an architecture by referring to the number of *hidden* layers; that is, layers that are neither inputs nor outputs. Thus, the network in the Figure is also referred to as a 2-hidden layer network. The multiple nodes in the output layer typically correspond to multiple classes for the multi-class pattern recognition problem.

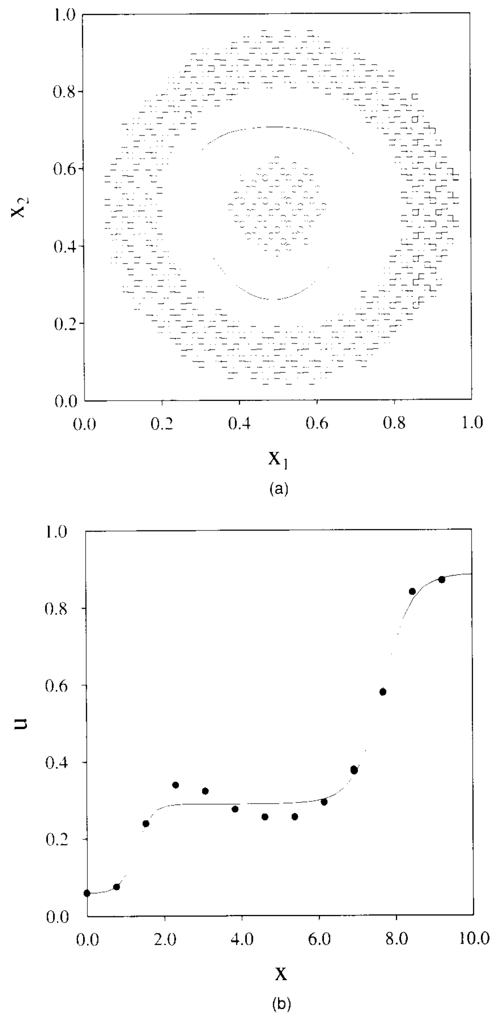
MLP Functional Capabilities

The capabilities of the MLP can be viewed from three different perspectives. The first has to do with its ability to implement Boolean logic functions, the second with its ability to partition the pattern space for classification problems, and the third with its ability to implement nonlinear transformations for functional approximation problems.

In the previous section we found that a single perceptron can implement only a vanishingly small fraction of the total number of logic functions of n variables. By cascading perceptrons together in layers however, arbitrary logic functions can be implemented.

Since it is possible to implement any logic function using two layers of fundamental operations (ANDs and ORs), no more than two layers of perceptrons are needed to implement an arbitrary logic function [95]. Some logic functions, however, may require an extremely large number of hidden layer nodes (at most exponential in n) [55, 96]. Finally, with the addition of feedback connections and a mechanism for implementing a unit time delay, it can be argued that perceptrons are capable of simulating a complete digital computer [131]. However, in neural networks one is not so much interested in implementing known logic functions as in *learning* the logic functions that describe a set of input/output examples.

Examples of the MLP's application to pattern classification and functional approximation are shown in Fig. 5. In both cases a 2-layer network has been used. In Fig. 5a there are three nodes in the hidden layer, and in Fig. 5b there are two. In the classification problem, the nonlinear decision boundary which separates the inner class from the outer class is a combination of three linear boundaries, one from each of the three hidden layer nodes. The linear boundaries are connected in a smooth fashion because of the sigmoid nonlinearity. In the interpolation problem, the function is approximated by a combination of two rounded step functions, one from each of the two nodes in the hidden layer. Again, the approximation is smooth and continuous because of the sigmoid function. This approximation can be made more accurate by including additional nodes in the hidden layer. In both of these examples we see that the MLP solves the problem by combining simple functional units formed by the hidden layer nodes. In the classification problem these units are hyperplanes, and in the functional approximation problem they are rounded step functions.



5. *Examples using a 2-layer perceptron network. Classification (a); functional approximation (b).*

For classification problems, Lippmann demonstrated that a 2-layer MLP can implement arbitrary convex decision boundaries [81]. Later it was shown that a 2-layer network can form an arbitrarily close approximation to any nonlinear decision boundary [84]. It has also been shown that a 2-layer MLP is capable of forming an arbitrarily close approximation to any continuous nonlinear mapping [24]. However, these results do not necessarily imply that there is no benefit to having more than two layers. For some problems, a small 3-layer network can be used where a 2-layer network would require an infinite number of nodes [21]. It has also been shown that there are problems which require an exponential number of nodes in a 2-layer network that can be implemented with a polynomial number of nodes in a 3-layer network [42], although it is an open problem whether a similar statement could be made for networks with more than three layers. None of these results require the use of the sigmoid nonlinearity in Eq. 2. The proofs in these papers assume only that the nonlinearity is a continuous smooth monotonically increasing function that is bounded above and below. Thus, numerous alternatives to the sigmoid have been proposed. Examples include $tanh(\alpha)$, $\mathrm{erf}(\alpha)$, and $\alpha/(1+|\alpha|)$. In addition, none of the above results require that the nonlinearity be present at the output layer. Thus, it is quite common to use linear output nodes since this tends to make learning easier.

MLP Learning Algorithm (Backpropagation)

One of the limitations of Rosenblatt's original formulation of the MLP was the lack of an adequate learning algorithm [89]. Algorithms were eventually developed to overcome this limitation [106, 119, 144]. The most common approach is to use a gradient descent algorithm, but the key difficulty in deriving such an algorithm for the MLP was that the gradient is zero almost everywhere when the hard-limiting nonlinearity is used. The nonlinearity must be present however, because without it the MLP would implement nothing more than a linear transformation at each layer, in which case the MLP could be reduced to an equivalent single layer network. The solution is to use a nonlinearity that is differentiable. The nonlinearity most often used is the sigmoid function. With this nonlinearity, it is possible to implement a gradient search of the weight space. Before we derive the learning algorithm, let us introduce the following notation:

Notation	Meaning
$u_{l,j}$	output of the jth node in layer l
$w_{l,j,i}$	weight which connects the ith node in layer l-1 to the jth node in layer l
$\mathbf{x}_p$	pth training sample
$u_{0,i}$	ith component of the input vector
$d_j(\mathbf{x_p})$	desired response of the jth output node for the pth training sample
N_l	number of nodes in layer l
L	number of layers
P	number of training patterns

For notational convenience we let the 0th layer of the network hold the input vector components; that is, in our notation $u_{0,j} = x_j$, where x_j is the jth component of the current input vector. Also, in order to account for the bias weights we define the 0th component of the input vector to each layer to be equal to 1; that is $u_{l,0} = 1$, i.e., $w_{l,j,0}$ are the bias weights. With this understanding, the output of a node in layer l is given by:

$$u_{l,j} = f\left(\sum_{i=0}^{N_{l-1}} w_{l,j,i}\, u_{l-1,i} \right) \tag{5}$$

where f(·) is the sigmoid nonlinearity. This function has a simple derivative:

$$f'(\alpha) = \frac{df(\alpha)}{d\alpha} = f(\alpha)\,(1-f(\alpha)) \tag{6}$$

The most common learning algorithm for the MLP uses a gradient search technique to find the network weights that minimize a criterion function. The criterion function to be minimized is the Sum-of-Squared-Error Criterion function:

$$J(\boldsymbol{w}) = \sum_{p=1}^{P} J_p(\boldsymbol{w}) \tag{7}$$

where P is the number of training patterns, $J_p(\boldsymbol{w})$ is the total squared error for the pth pattern:

$$J_p(\boldsymbol{w}) = \frac{1}{2} \sum_{q=1}^{N_L} (u_{L,q}(\boldsymbol{x_p}) - d_q(\boldsymbol{x_p}))^2 \tag{8}$$

and N_L is the number of nodes in the output layer as defined above. The weights of the network are determined iteratively according to:

$$\begin{aligned} w_{l,j,i}(k+1) &= w_{l,j,i}(k) - \mu \left.\frac{\partial J(\boldsymbol{w})}{\partial w_{l,j,i}}\right|_{\boldsymbol{w}(k)} \\ &= w_{l,j,i}(k) - \mu \sum_{p=1}^{P} \left.\frac{\partial J_p(\boldsymbol{w})}{\partial w_{l,j,i}}\right|_{\boldsymbol{w}(k)} \end{aligned} \tag{9}$$

where μ is a positive constant called the *learning rate*. To implement this algorithm we must develop an expression for the partial derivative of J_p with respect to each weight in the network. For an arbitrary weight in layer l this can be written using the Chain Rule:

$$\frac{\partial J_p(\boldsymbol{w})}{\partial w_{l,j,i}} = \frac{\partial J_p(\boldsymbol{w})}{\partial u_{l,j}} \frac{\partial u_{l,j}}{\partial w_{l,j,i}} \tag{10}$$

where:

$$\frac{\partial u_{l,j}}{\partial w_{l,j,i}} = \frac{\partial}{\partial w_{l,j,i}} \left[f\left(\sum_{m=0}^{N_{l-1}} w_{l,j,m}\, u_{l-1,m} \right) \right] \tag{11}$$

$$= f'\Big(\sum_{m=0}^{N_{l-1}} w_{l,j,m}\, u_{l-1,m}\Big) \frac{\partial}{\partial w_{l,j,i}} \left[\sum_{m=0}^{N_{l-1}} w_{l,j,m}\, u_{l-1,m} \right]$$

$$= f'\Big(\sum_{m=0}^{N_{l-1}} w_{l,j,m}\, u_{l-1,m}\Big)\, u_{l-1,i}$$

Substituting from Eq. 6 for the first term, we get:

$$\frac{\partial u_{l,j}}{\partial w_{l,j,i}} = u_{l,j}(1 - u_{l,j})\, u_{l-1,i} \tag{12}$$

With this, Eq. 10 becomes:

$$\frac{\partial J_p(\mathbf{w})}{\partial w_{l,j,i}} = \frac{\partial J_p(\mathbf{w})}{\partial u_{l,j}}\, u_{l,j}\,(1 - u_{l,j})\, u_{l-1,i} \tag{13}$$

The term $\partial J_p(\mathbf{w})/\partial u_{l,j}$ represents the *sensitivity* of $J_p(\mathbf{w})$ to the output of node $u_{l,j}$. The node $u_{l,j}$ exhibits its influence on J_p through all of the nodes in the succeeding layers. Thus, $\partial J_p(\mathbf{w})/\partial u_{l,j}$ can be expressed as a function of the sensitivities to nodes in the next highest layer as follows:

$$\frac{\partial J_p(\mathbf{w})}{\partial u_{l,j}} = \sum_{m=1}^{N_{l+1}} \frac{\partial J_p(\mathbf{w})}{\partial u_{l+1,m}} \frac{\partial u_{l+1,m}}{\partial u_{l,j}} \tag{14}$$

$$= \sum_{m=1}^{N_{l+1}} \frac{\partial J_p(\mathbf{w})}{\partial u_{l+1,m}} \frac{\partial}{\partial u_{l,j}} \left[f\Big(\sum_{q=0}^{N_l} w_{l+1,m,q}\, u_{l,q}\Big) \right]$$

$$= \sum_{m=1}^{N_{l+1}} \frac{\partial J_p(\mathbf{w})}{\partial u_{l+1,m}} f'\Big(\sum_{q=0}^{N_l} w_{l+1,m,q}\, u_{l,q}\Big) \frac{\partial}{\partial u_{l,j}} \left[\sum_{q=0}^{N_l} w_{l+1,m,q}\, u_{l,q} \right]$$

$$= \sum_{m=1}^{N_{l+1}} \frac{\partial J_P(\mathbf{w})}{\partial u_{l+1,m}}\, u_{l+1,m}\,(1 - u_{l+1,m})\, w_{l+1,m,j}$$

This process can be continued for $\partial J_p(\mathbf{w})/\partial u_{l+1,m}$ and so on, until we reach the output layer. At the output layer we reach a *boundary condition* where the sensitivities of the nodes in the last layer are derived from Eq. 8 as:

$$\frac{\partial J_p(\mathbf{w})}{\partial u_{L,j}} = u_{L,j}(\mathbf{x}_p) - d_j(\mathbf{x}_p) \tag{15}$$

While the derivation seems to be working its way forward to the output layer, the sensitivity of a node is actually computed from the output layer *backwards*. The expression in Eq. 15 is called the output error, and the corresponding expression for hidden layer nodes in Eq. 14 is often referred to as the hidden layer error, although strictly speaking it doesn't represent an error of any type. Since this "hidden layer error" is computed from the output layer backwards, it has historically been called the *backpropagated error*, and the learning algorithm the *BackpropagationAlgorithm*. The results in Eqs. 13-15 can be used in Eq. 9 to implement the gradient search. Typically the summation in Eq. 9 is replaced with an estimate of the gradient based on a single sample. That is, typically $\partial J(\mathbf{w})/\partial w_{l,j,i}$ is approximated so that Eq. 9 becomes:

$$w_{l,j,i}(k+1) = w_{l,j,i}(k) - \mu \left. \frac{\partial J_{k \bmod P}(\mathbf{w})}{\partial w_{l,j,i}} \right|_{\mathbf{w}(k)} \tag{16}$$

where *(k mod P)* is the index of the pattern used to estimate the gradient at the kth iteration.

Equations 13-16 comprise the Backpropagation learning algorithm. The complete algorithm is shown in Table 1. The weights are typically initialized to small random values. This starts the search in a relatively "safe" position [58]. The learning rates can be chosen in a number of different ways. They can be the same for every weight in the network, different for each layer, different for each node, or different for each weight in the network. In general it is difficult to determine the best learning rate, but a useful rule of thumb is to make the learning rate for each node inversely proportional to the average magnitude of vectors feeding into the node. Several attempts have been made to adapt the learning rate as a function of the local curvature of the surface [11, 25, 63, 119]. The simplest approach, and one that works quite well in practice, is to add a *momentum* term of the form $\alpha(w_{l,j,i}(k) - w_{l,j,i}(k{-}1))$ to each weight update, where $0 < \alpha < 1$. This term makes the current search direction an exponentially weighted average of past directions, and helps keep the weights moving across flat portions of the performance surface after they have descended from the steep portions.

The process of computing the gradient and adjusting the weights is repeated until a minimum (or a point sufficiently close to the minimum) is found. In practice it may be difficult to automate the termination of the algorithm. However, if one wishes to do so, there are several stopping criteria that can be considered. The first is based on the magnitude of the gradient. The algorithm can be terminated when the magnitude of the gradient is sufficiently small, since by definition the gradient will be zero at the minimum.

Second, one might consider stopping the algorithm when J falls below a fixed threshold. However this requires some knowledge of the minimal value of J, which is not always available. In pattern recognition problems, one might consider stopping as soon as all of the training data are correctly classified. This assumes, however, that the network can actually classify all of the data correctly, which won't always be the case. Even if it can do so, this stopping criterion may not yield a solution that generalizes well to new data.

Third, one might consider stopping when a fixed number of iterations have been performed, although there is little guarantee that this stopping condition will terminate the algorithm at a minimum. Finally, the method of *cross-validation* can be used to monitor generalization performance during learning, and terminate the algorithm when there is no longer an improvement. The method of cross-validation works by splitting the data into two sets: a *training set* which

```
procedure BACK_PROP
  Initialize the weights to small random values ;
  repeat
    Choose next training pair (x,d) and let the $0^{th}$ layer be $\mathbf{u}_0 = \mathrm{x}$ ;
    FEED_FORWARD ;
    COMPUTE_GRADIENT ;
    UPDATE_WEIGHTS ;
  until termination condition reached ;
end ; {BACK_PROP}

subroutine FEED_FORWARD
  for $layer$ = 1 to $L$ do
    for node = 1 to $N_{layer}$ do $u_{layer,node} = f(\sum_{i=0}^{N_{layer-1}} w_{layer,node,i} u_{layer-1,i})$ ;
    endloop
  endloop
end ; {FEED_FORWARD}

subroutine COMPUTE_GRADIENT
  for $layer = L$ to 1 do
    for $node$ = 1 to $N_{layer}$ do
      if $layer = L$ then $e_{L,node} = u_{L,node} - \mathrm{d}_{node}$ ;
      else $e_{layer,node} = \sum_{m=1}^{N_{layer-1}} e_{layer+1,m} u_{layer+1,m}(1 - u_{layer+1,m}) w_{layer+1,m,node}$ ;
    endloop
    for all weights in layer $layer$ do
      $g_{layer,j,i} = e_{layer,j} u_{layer,j}(1 - u_{layer,j}) u_{layer-1,i}$ ;
    endloop
  endloop
end ; {COMPUTE_GRADIENT}

subroutine UPDATE_WEIGHTS
  for all $w_{l,j,i}$ do
    $w_{l,j,i}(k+1) = w_{l,j,i}(k) - \mu g_{l,j,i}$ ;
  endloop
end ; {UPDATE_WEIGHTS}
```

Table 1: Backpropagation Learning Algorithm

is used to train the network, and a *test set* which is used to measure the generalization performance of the network. During learning, the performance of the network on the training data will continue to improve, but its performance on the test data will only improve to a point, beyond which it will start to degrade. It is at this point, where the network starts to *overfit* the training data, that the learning algorithm is terminated.

The first three criteria are sensitive to the choice of parameters, and if not chosen properly the results can be very poor due to premature termination. Cross-validation, however, does not suffer from this characteristic. It not only avoids premature termination, but can actually improve the generalization performance of the network. However, cross-validation can be more computationally intensive. Further, if the number of data samples is limited, cross-validation reduces the size of the training set even further.

MLP example

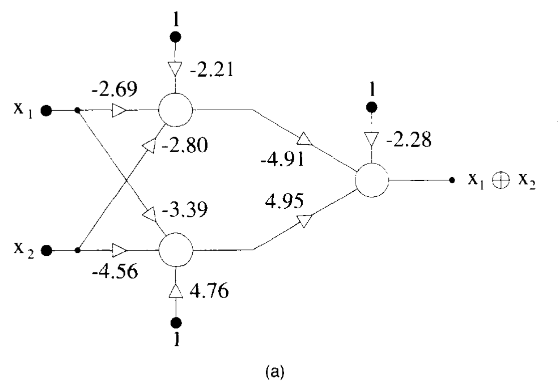
Examples of how the MLP can be used for classification and functional approximation were shown previously in Fig. 5. In both of these examples, a 2-layer network was trained using the Backpropagation algorithm. We now consider an example where the MLP is trained to implement a simple logic function.

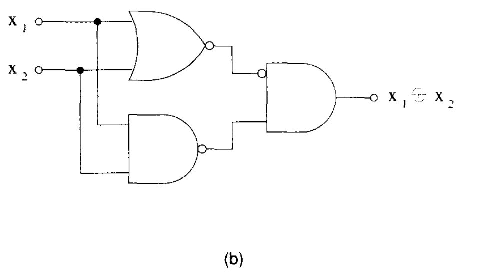
Consider the 2-input XOR problem. This problem has a great deal of historical significance because it is one of the simplest logic functions that *cannot* be implemented by a single perceptron; that is, it requires at least two layers of perceptrons [89]. This problem has four training patterns corresponding to the four combinations of two bipolar inputs, {(-1, -1), (-1, 1), (1, -1), (1, 1)}. The objective is to teach an MLP to respond with a high output when the two inputs are different and a low output when they are the same.

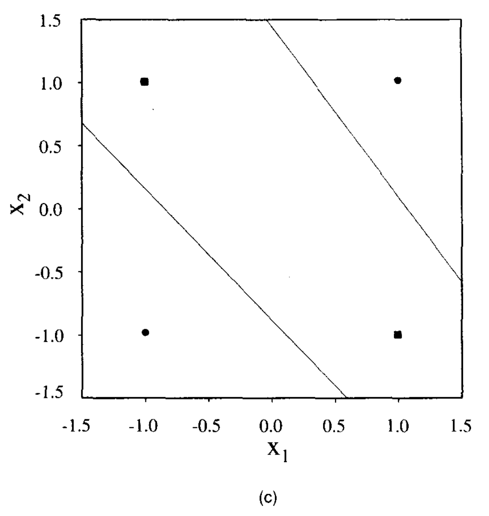
Output target values of 0 and 1 are used in the training process. A 2-layer MLP is used with two hidden layer nodes and one output node. The weights of the network are initialized to small random values. In this example we used random numbers generated from a uniform distribution over the interval (-0.1, 0.1). The four patterns above were presented in mixed order so that the desired output alternates between 0 and 1 at each iteration. A fixed learning rate of μ=1.0 was used for all nine weights. After 4000 iterations (1000 cycles through the four training patterns), the Backpropagation algorithm found a weight solution which produced the results shown in Fig. 6. This algorithm can take varying amounts of time to solve this problem depending on the exact values of the initial weights and the learning rates that are used. In Fig. 6, we give three different representations of the solution: the weights learned by the network, a logic diagram equivalent, and the decision boundaries formed in the pattern space. It should be clear that there are many possible solutions to this problem, and that in this example the network has correctly learned one of these.

MLP Temporal Difference Learning

Neural networks are often used to predict the outcome of a future event based on current observations of the state of the environment. An example of one such use is in "game playing" where the network is asked to predict whether or not a particular board position will lead to a *win* at the end of the game. Given a choice of several moves, we can use the







6. *A solution to the XOR problem. Weight solution (a); logic diagram equivalent (b); partitioning of the pattern space (c).*

network to choose the move which arrives at the board position that is most likely to lead to victory. This type of application adds an interesting twist to the learning problem.

Standard learning algorithms like backpropagation cannot be used to update the weights after every move, since the outcome (the value that we want to predict) is not available until the end of the game. While it is possible to wait until the end of the game, and then update the weights (using information that was stored along the way), this would result in rather large storage requirements. *Temporal difference learning* is a technique developed by Sutton [132] for accumulating weight update information at each move along the way without having to save the states of the network and without having to wait until the end of the game for the final outcome.

Formally, we seek to minimize the following criterion function:

$$J(\boldsymbol{w}) = \sum_{p=1}^{P} \sum_{k=1}^{N_p} \lambda^{N_p - k} (z_{N_p} - G(\boldsymbol{x}_p(k)))^2 \tag{17}$$

where P is the number of examples, e.g., the number of games; N_p is the number of steps in the pth example, e.g., the length of the game, which is not known until the outcome is determined; z_{N_p} is the actual outcome for the pth example; $G(x_p(k))$ is the output of the network when presented with $x_p(k)$, and λ «(0,1) is a parameter which is used to place more emphasis on predictions temporally close to the outcome. The value of $G(\boldsymbol{x}_p(k))$ is the predicted outcome based on the observation $\boldsymbol{x}_p(t)$, and can correspond to the output of a simple linear function or something more complex such as an MLP. For example, $\boldsymbol{x}_p(t)$ might be a description of the current board position in a game of backgammon and $\boldsymbol{x}_p(k)$ would be the predicted outcome of the game based on that board position.

As usual, we update the weights of our network using a gradient descent learning algorithm. However, as mentioned above, a direct application of this type of algorithm is not particularly amenable to an on-line approach. The problem is that we do not know the outcome z_{N_p} until we have gone through the complete set of N_p observations for a particular example. A straightforward approach to learning can result in large storage requirements since we must maintain copies of the states of the network at each time, k, in order to implement a gradient descent algorithm such as backpropagation. Moreover, most of our computations cannot be performed until the final time step N_p when the outcome, z_{N_p} is observed. Sutton has developed an algorithm for implementing these prediction problems that is suitable for on-line computation [132]. Specifically, the algorithm does not require the states of the network to be stored for each time step, and further it distributes the computation of the gradient uniformly over the entire time sequence. While the derivation of the algorithm is relatively straightforward, it is beyond the scope of this paper. The final algorithm is shown in Table 2. This algorithm is commonly known as the *TD* (λ) algorithm, or the *Temporal Difference Learning Algorithm*. Note this algorithm is slightly different than Sutton's algorithm in that λ would be equal to 1 in equations (A) and (B).

The *TD* (λ) algorithm has been used successfully as an approach for learning the game of backgammon from the outcome of self-play [133]. Specifically, the algorithm attempts to learn the evaluation function that determines which

```
procedure $TD(\lambda)$
  Initialize network ;
  repeat
    Begin next example (e.g. start next game) ;
    $e_{0,i} = 0, \gamma_{0,i} = 0, k = 1$ ;
    repeat
      Get next observation $x_p(k)$ ;
      Calculate $G(x_p(k))$ ;
      for all $w_i$ do
        $e_{k,i} = \frac{\partial G(x_p(k))}{\partial w_i} + \lambda e_{k-1,i}$;
        $\gamma_{k,i} = [G(x_p(k)) - G(x_p(k-1))]e_{k-1,i} + \lambda \gamma_{k-1,i}$;        (A)
        $k = k + 1$ ;
      endloop
    until outcome determined (Note: $k = N_p$) ;
    for all $w_i$ do
        $\gamma_{k,i} = [z_{N_p} - G(X_p(k))]e_{k-1,i} + \lambda \gamma_{k-1,i}$;
      $w_i(m+1) = w_i(m) - \mu \gamma_{k,i}$ ;                                          (B)
    endloop
  until termination condition reached ;
end ;   {$TD(\lambda)$}
```

Table 2: Temporal Difference Learning Algorithm

board positions are more likely to lead to a win. The resulting program has reached near-expert level performance and has achieved good results against human Grand Masters. In fact, the resulting network was capable of beating two-time world champion Bill Robertie 13 out of 31 times, and has been shown to win approximately 60 percent of the games against Neurogammon 1.0, winner of the 1989 Computer Olympiad.

MLP Issues and Limitations

The MLP is capable of approximating arbitrary nonlinear mappings, and given a set of examples, the Backpropagation algorithm can be called upon to learn the mapping at the example points. However, there are a number of practical concerns. The first is the matter of choosing the network size. The second is the time complexity of learning. That is, we may ask if it is possible to learn the desired mapping in a reasonable amount of time. Finally, we are concerned with the ability of our network to generalize; that is, its ability to produce accurate results on new samples outside the training set.

Choosing the Network Size

First, theoretical results indicate that the MLP is capable of forming arbitrarily close approximations to any continuous nonlinear mapping; but this is true only as the size of the network grows arbitrarily large [24]. In general, it is not known what (finite) size network works best for a given problem. Further, it is not likely that this issue will be resolved in the general case since each problem will demand different capabilities from the network. Choosing the proper network size is important. If the network is too small, it will not be capable of forming a good model of the problem. On the other hand, if the network is too big then it may be *too capable* [10]. That is, it may be able to implement *numerous solutions* that are consistent with the training data, but most of these are likely to be poor approximations to the actual problem. In this case, the solution learned during any given training session is likely be a poor approximation to the actual problem.

Ultimately, we would like to find a network whose size best matches the capability of the network to the structure of underlying problem; or, since the data is sometimes not sufficient to describe all of the intricacies of the underlying problem, we would like a network whose size best captures the structure of the data. With some specific knowledge about the structure of the problem (or data), and a fundamental understanding of how the MLP might go about implementing this structure, one can sometimes form a good estimate of the proper network size.

The examples in Fig. 5 provide some insight into how a 2-layer MLP might go about solving a pattern recognition problem (by piecing together linear boundaries in a smooth fashion) and a functional approximation problem (by combining smooth step functions).

With little or no prior knowledge of the problem however, one must determine the network size by trial and error. A methodical procedure is recommended. One approach is to start with the smallest possible network and gradually increase the size until the performance begins to level off. In this approach, each size network is trained independently. A closely related approach is to "grow a network." The idea here is to start with one node and create additional nodes as they are needed. Approaches that use such a technique include Cascade Correlation [30], the Group Method of Data Handling (GMDH) [8, 62], Projection Pursuit [33, 34], the Algorithm for Synthesis of Polynomial Networks (ASPN) [8], and others [9]. These approaches differ from the previous approach in that additional nodes are created *during the training process*.

Another possibility is to start with a large network and then apply a pruning technique that destroys weights and/or nodes which end up contributing little or nothing to the solution [77]. With this approach one must have some idea of what size network constitutes a "large" network; that is, "what size network is probably too big for the problem?"

The following guidelines are useful in placing an upper bound on the network size. For a fully connected MLP network, no more than three layers are typically used, and in most cases only two. Numerous bounds exist on the number of hidden layer nodes needed in 2-layer networks. For example, it has been shown that an upper bound on the number of hidden layer nodes needed for the MLP to implement the training data exactly is on the order of P, the number of training samples [55]. This suggests that one should never use more hidden layer nodes than training samples. Actually, the number of hidden layer nodes should almost always be much less than the number of training samples, otherwise the network simply "memorizes" the training samples, resulting in poor generalization.

It is more useful to consider bounds on the number of hidden layer nodes that are expressed as a function of n, the dimension of the input pattern. It is possible to find problems that require the number of hidden layers nodes to be exponential in n, but it is generally recommended that MLPs be used for problems that require no more than a polynomial number of hidden layer nodes [26, 42, 127]. For example, if the problem is one that requires a pattern class to be completely enclosed by a spherical decision boundary, then the number of hidden layer nodes should be approximately $3n$ [59].

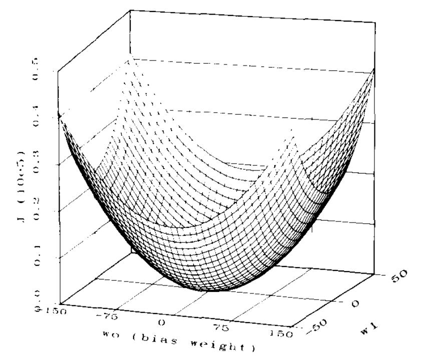
Complexity of Learning

Even if one is able to determine the optimal network size, it turns out that finding the correct weights for a network is an inherently difficult problem. The problem of finding a set of weights for a fixed-size network which performs the desired mapping exactly for some training set is known as the *loading problem*. Recently it has been shown that the loading problem is NP-complete [13, 65]. This suggests that if we have a very large problem, e.g., if the dimension of the input space is very large, then it is unlikely that we will be able to determine if a weight solution exists in a reasonable amount of time.

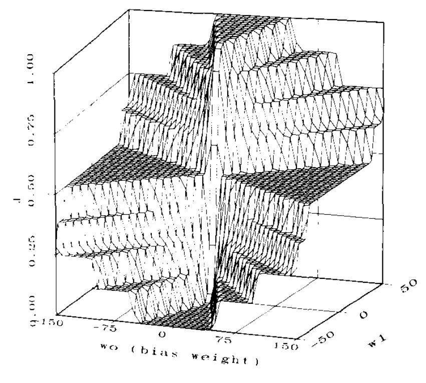
On the other hand, learning algorithms like Backpropagation are based on a gradient search, which is a greedy algorithm that seeks out a local minimum and thus may not yield the exact mapping. Gradient search algorithms usually don't take exponential time to run (if a suitable stopping criterion is employed). However, it is well known that finding local weight solutions using Backpropagation is extraordinarily slow. One way to explain this sluggishness is to characterize the error surface which is being searched. In the case of a single perceptron with linear activation function the error surface is a quadratic bowl (with a single (global) minimum), and thus it is a relatively agreeable surface to search. For MLPs however, it turns out that the surface is quite harsh [58]. This is illustrated in Fig. 7, which shows J for two different cases. In both cases there is a single node with two weights and the training samples are {-4, -3, -2, -1} for class 1 and {1, 2, 3, 4} for class 2. The only difference between the two cases is that in Fig. 7a the node is linear, and in Fig. 7b a sigmoid is used. The surface in Fig. 7b is for a single node only, but it has been shown that many of the characteristics of this surface are typical of surfaces for multilayer perceptrons as well [58].

These surfaces tend to have a large amount of flatness as well as extreme steepness, but not much in between. It is difficult to determine if the search has even terminated with this type of surface since the transient flat spots "look" much the same as minima, i.e., the gradient is very small. Furthermore, with this type of surface a gradient search moves very slowly along these flat parts.

It is dangerous to increase the learning rate to compensate for the sluggishness in these areas because the algorithm may then exhibit instabilities when it reaches the steep parts of the surface. Attempts to speed learning include variations on simple gradient search [63, 110, 119], line search methods [60], and second-order methods [12, 140]. Although most of these have been somewhat successful, they usually introduce additional parameters which are difficult to determine, must be varied from one problem to the next, and if not chosen properly can actually slow the rate of convergence.



(a)



(b)

7. *Error surface examples. J for a linear node (a); J for a nonlinear node with a sigmoid (b).*

Generalization

Generalization is a measure of how well the network performs on the actual problem once training is complete. It is usually tested by evaluating the performance of the network on new data outside the training set. Generalization is most heavily influenced by three parameters: the number of data samples (and how well they represent the problem at hand), the complexity of the underlying problem, and the network size. Generally speaking, a larger number of data samples will do a better job at representing the underlying problem and, as long as the proper network size is used, this should allow us to learn a better solution to the problem.

The generalization issue is often viewed from two different perspectives. In the first, the size of the network is fixed (presumably in accordance with the complexity of the underlying problem) and the issue becomes: "How many training samples are required to achieve good generalization?" This perspective is useful in applications where we have the ability to acquire as many samples as we deem necessary. In the second case, the number of training samples is fixed and the issue becomes: "What size network gives the best generalization for this data?" This perspective is useful in applications where we are limited in our ability to acquire training data, and we would like to know what size network is best at describing the data we have available. Both are valid viewpoints, although the first is probably more common in the theoretical literature, and will therefore be the one adopted here.

Although the concepts discussed in this section are applicable to the general learning problem, they are perhaps most easily understood in the context of a specific problem; namely that of learning logic functions. We will focus our discussion on the problem of learning logic functions of d variables. This problem has a finite domain (2^d patterns) and a range of $\{0,1\}$.

We can think of a network as a device that formulates *hypotheses* as to what the true logic function might be: the larger the network, the larger the set of functions it can form, and the more likely the true logic function is in this set. Further, we can view the training samples as *clues* that help us discover the correct function.

Training samples allow us to reject logic functions which are not consistent with the clues. We assume for now that the network size is chosen sufficiently large to include the true logic function in its set of implementable functions. If it is not, then the solution we seek is the one (or more) function(s) that best approximates the true logic function.

In a sense, the learning process can be viewed as a process of elimination: the more training data we have, the more incorrect functions we are able to reject, and the more likely we are to find the correct function.

Given this perspective, there are two different approaches for studying generalization. The first attempts to determine the *average (or expected) generalization* of the network, while the second attempts to *bound the worst-case generalization.*

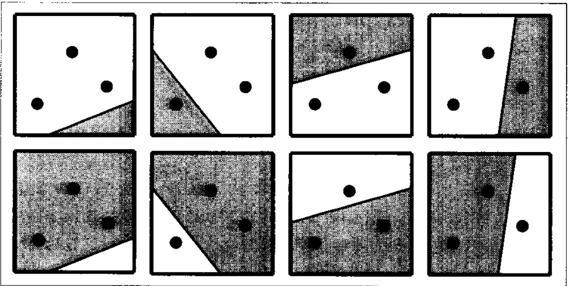
The average generalization of the network is determined by allowing Q to represent the set of all logic functions implementable by the network. Associated with each function, $q \in Q$, is a measure of its generalization, $g(q)$. Typically $g(q)$ is taken to be the fraction of the domain, i.e., the fraction of all 2^d samples, for which q produces the correct output. Now suppose that the network is presented with P training samples. Let $Q_P \subset Q$ be the subset of functions in Q that are consistent with the set of P training samples. Then the average (or expected) generalization of the network is found by averaging the generalization over all functions in Q_P. Additional data samples serve to improve generalization by reducing the size of Q_P while at the same time retaining the functions that are consistent with the new data. Although very interesting, this approach can be difficult to apply in practice. The details of this approach are beyond the scope of this article [26, 48, 83, 93].

The second approach for studying generalization uses a worst-case analysis to compute a bound on the *generalization error.* The generalization error is defined to be the difference between the generalization on the training samples and the generalization on the actual problem. We expect the generalization on the training data to be overly optimistic since the learning algorithm concentrates its efforts on this data. In many cases however, the difference between this estimate and the actual generalization can be bounded, and by increasing the number of training samples this bound can be made arbitrarily small. This was shown by Vapnik and Chervonenkis [135]. A key result of their work was that a useful bound can only be established when the number of training samples exceeds a parameter called the *Vapnik-Chervonenkis Dimension* (VCdim), which is a measure of the capability of the system. In our case, the system is an MLP. Formally the VCdim is defined as follows:

The VCdim of a system is the size of the largest set S of data samples for which the system can implement all possible $2^{|S|}$ *dichotomies on S*

where $|S|$ is used to denote the cardinality of S; that is, the number of samples in S. Note this definition requires only that one such set of examples exist, it need not be true for all sets of size VCdim. As an example, the VCdim of a single perceptron with a two-dimensional input is 3. This is easily verified since it is possible to find a set of 3 points that can be linearly dichotomized (partitioned into two groups with a linear boundary) in all 2^3 ways (Fig. 8), and at the same time there is no set of 4 points that can be linearly dichotomized in all 2^4 ways. In general, it can be shown that the VCdim of a single perceptron with a n-dimensional input is $n + 1$.

It is possible for the VCdim of a system to be infinite, in which case it becomes impossible to bound the worst case generalization error for the system (although it may still be possible to learn solutions that exhibit some degree of generalization on average). Fortunately the VCdim for the


8. *Eight possible ways to dichotomize three points in a plane.*

MLP has been shown to be finite. For example, the VCdim of a one-hidden-layer MLP with complete connections between the layers has been shown [10] to lie in the range:

$$2\left[\frac{N_1}{2}\right] n \le VCdim \le 2N_w \log(eN_N) \tag{18}$$

where $[\cdot]$ is the *floor* operator that returns the largest integer less than its argument, N_l is the number of hidden layer nodes, n is the dimension of the input pattern, N_w is the total number of weights in the network, e is the base of the natural logarithm, and N_N is the total number of nodes in the network. The lower bound is approximately equal to the number of weights connecting the input to the hidden layer, which usually accounts for a majority of the weights the network. The upper bound is not much more than twice the number of weights in the network. Thus, as a rule of thumb, we can use the number of weights in the network as a rough estimate of the true VCdim. These results assume that all nodes in the network use hard-limiting nonlinearities. It is more difficult to determine the VCdim when sigmoids are used, but Sontag's results suggest that it is at least twice as large in this case [128]. Also, only the lower bound above assumes a one-hidden-layer network. The upper bound holds regardless of the number of layers and the connectivity.

Once the VCdim of a system is known, it is possible to determine the number of training samples required for good generalization. Vapnik and Chervonenkis's result suggests that the number of samples be larger than the VCdim. A useful rule of thumb is that the number of training samples be approximately ten times the VCdim. For an MLP this means that the number of training samples should be approximately ten times the number of weights. This rule seems to work fairly well in practice [148]. In addition, there are more exact expressions relating the VCdim and the number of training samples [10, 100].

The above result has strong implications in practice since the number of weights can grow quite large in many problems. Consider, for example, the handwritten character recognition problem [76] in which the network must learn to discriminate between the handwritten digits "0" through "9". Each digit is represented as a 16 by 16 binary image. Thus, the dimension of the input to the MLP is 256. A fully connected 2-layer network with 12 hidden layer nodes and 10 output layer nodes (one for each digit class) would have a total of 3214 weights [75]. Using the rule of thumb mentioned above, we would need approximately 32,140 examples of handwritten characters to expect good generalization from the network. In practice it may be difficult to obtain this many samples. In addition, during training we would probably have to cycle through this training set several times before the algorithm converges to a solution. The training time required for the Backpropagation algorithm on this data would probably be on the order of several days using a fast single-CPU machine.

Thus, a large number of weights adversely affects not only generalization, but also the time required to learn the solution. It is therefore to our advantage to seek methods for reducing the number of weights, while at the same time retaining the capability of solving the problem. The next three sections are devoted to such methods.

Improving MLP Generalization Through Pruning

It is generally true that there is a large amount of redundant information contained in the weights of a fully connected MLP. Thus it seems plausible that we could eliminate weights from the network, and at the same time retain the functional capability needed to solve the problem. This process is known as *pruning*. There are two advantages to pruning. First, with a fixed number of training samples the reduction in weights can lead to a marked improvement in the generalization properties of the network. Second, by isolating the relevant parameters, learning is presumably easier.

The simplest approach to pruning is to delete the smallest weights in the network. This however is not always the best approach since the solution can be quite sensitive to these weights. A much better approach is to delete the weights that end up disturbing the solution the least.

A very popular method based on this approach is called *Optimal Brain Damage* (OBD) [77]. In OBD, the network is first trained using the Backpropagation algorithm. Then the weights with the smallest *saliency* are deleted. After this, the reduced size network is re-trained to obtain the final solution (it may be useful to repeat this procedure of deleting weights and re-training the network several times before settling on a final solution). The saliency for weight $w_{l,i,j}$ is defined as:

$$s_{l,i,j} = \delta J_{l,i,j}\, w^2_{l,i,j} \tag{19}$$

where $\delta J_{l,i,j}$ measures the sensitivity of the criterion function to small perturbations in wl,i,j. This procedure avoids the deletion of small weights which have a large influence on the solution because the saliency value for these weights is inflated by $\delta J_{l,i,j}$. In OBD, the sensitivity measure is approximately:

$$\delta J_{l,i,j} \approx \frac{\partial^2 J}{\partial w^2_{l,i,j}} = \sum_{p=1}^{P} \frac{\partial^2 J_p}{\partial w^2_{l,i,j}} = \sum_{p=1}^{P} r_{l,i}\, u^2_{l-1,j} \tag{20}$$

This approximation is obtained by expanding δJ in a Taylor series and dropping all second-order cross terms as well as all higher-order terms. The first-order terms drop out automatically because the network has converged to a local minimum where first-order terms are zero. To implement the above equation we need an expression for $r_{l,i}$. For hidden layer nodes it is given by:

$$r_{l,i} = (f'(y_{l,i}))^2 \left[\sum_{m=1}^{N_l} r_{l+1,m} w^2_{l+1,m,i}\right] + f''(y_{l,i}) \frac{\partial J}{\partial u_{l,i}} \tag{21}$$

where f'' is the second derivative of the nonlinear activation function, $\partial J/\partial u_{l,i}$ is defined in Eq. 14, and $y_{l,i}$ is the weighted sum for node l,i:

$$y_{l,i} = \sum_{m=1}^{N_{l-1}} w_{l,i,m}\, u_{l-1,m} \tag{22}$$

At the output layer, $r_{L,i}$ is given by:

$$r_{L,i} = (f'(y_{L,i}))^2 - (d_j - u_{L,j})f''(y_{L,i}) \tag{23}$$

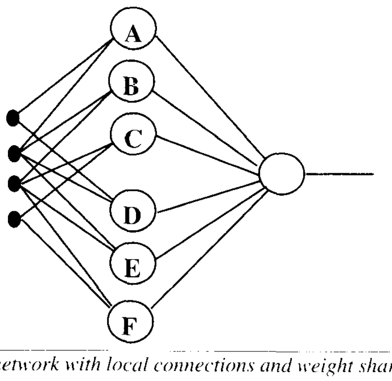
In summary, the OBD method comprises the following steps. First, the network is trained with the Backpropagation algorithm. The saliencies are then computed using the above equations. The weights with the smallest saliency are deleted, and the network is retrained to obtain the final solution. In some applications it has been found that the number of weights can be reduced by a factor of four [77]. A recent extension of the OBD method which uses the full Jacobian (instead of just the diagonal elements) is described in [47].

Improving Generalization Through Weight Sharing

Another way to reduce the number of weights in the network is through the use of *local connections* and *weight sharing* [76, 138]. The basic idea behind local connections is to make individual nodes process only a local region of the input.

For example, in the handwritten character recognition problem, hidden layer nodes may process only a small m-by-m patch, e.g., m = 5, of the input pattern. Of course, this requires several hidden layer nodes to cover the entire input pattern. It is possible that multiple nodes may be used to cover the same region, and that adjacent regions may overlap.

This is illustrated in Fig. 9 for a very simple network, a 2-layer network with six hidden layer nodes. Each of the hidden layer nodes, A-F, is connected to a local region of the input. Adjacent regions overlap by one position. Nodes A, B, and C cover the same regions as D, E, and F, respectively. If this network were a fully connected MLP with six hidden layer nodes it would have a total of 37 weights. With the local connection scheme however, there are only 25 weights. On the other hand, a locally connected network may require more nodes than a fully connected network to solve the same problem. So even though local connections can reduce the number of weights per node, it may not always reduce the total number of weights, but when coupled with *weight sharing* a dramatic reduction can be achieved.



9. Simple network with local connections and weight sharing.

With weight sharing, nodes A, B, and C in Fig. 9 share all the same weights; similarly, so do D, E, and F. This reduces the total number of weights to 13. For large problems this approach can lead to a drastic reduction in the number of weights. Such an approach has been used successfully to solve the handwritten character recognition problem [75, 76]. In this problem, nodes with identical weights are positioned at every possible registration point in the input pattern [76]. Conceptually these nodes are all looking for the same m-by-m "feature," but each node is looking at a different registration point in the pattern. From another perspective, it is as if a single node with an m-by-m kernel is being used to scan the input pattern, producing as its output another pattern with one value for each registration point. Networks that use these techniques typically have more than one hidden layer of locally connected nodes with weight sharing. Excellent results have been reported using this technique with only four different groups of nodes in the first hidden layer, i.e., four different sets of weights, each shared by a group of nodes positioned at every registration point in the pattern, and twelve in the second hidden layer [76].

Improving Generalization
Through Complexity Regularization

Another way to improve generalization is through the use of *complexity regularization*. In this approach, a term is added to the criterion function that discourages the learning algorithm from seeking solutions which are too complex. This term then, represents a measure of the network's complexity, e.g., the number of weights. The resulting criterion function is of the form:

Cost = Mapping Error + Model Complexity

This type of criterion is sometimes referred to as a *Minimum Description Length* (MDL) criterion, because it is of the same form as the information theoretic measure of *description* length [115, 116]. Simply put, the description length of a set of data is the total number of bits required to represent the data. If a model, e.g., a neural network, is used to represent the data, then the total description length is the number of bits required to describe the model plus the number of bits required to encode the errors, i.e., the portion of the data which is not described by the model. The cost function above is of this form if we relate the average mapping error to the number of bits required to encode the errors, and the model complexity to the number of bits required to describe the model. The model that minimizes this cost function then, in some sense, provides a minimal description of the data. Cost functions of this type are used in the methods of *weight decay* and *weight elimination* discussed below.

Weight decay [43, 52] can be viewed as a way of reducing the *effective number of weights* in the network by encouraging the learning algorithm to seek solutions that use as many zero (or near zero) weights as possible. This is accomplished by adding a term to the criterion function that penalizes the network for using nonzero weights. The new criterion function takes on the form:

$$J(w) = \sum_{p=1}^{P} J_p(w) + \frac{\lambda}{2}\sum_i w_i^2 \tag{24}$$

where the first term is the same squared error criterion as before, and the second term is a new term involving a sum over all the weights in the network. The λ parameter is a small positive constant that is used to control the influence of this term relative to the squared error term. The learning algorithm derived from this criterion is a simple extension to the Backpropagation algorithm. In fact, the only modification that is needed is to subtract an extra term of the form $\mu\lambda w_i(k)$ each time the ith weight is updated.

The weight decay method does not actually delete weights from the network, nor does it typically produce weights that are exactly zero. This begs the question: How does weight decay actually improve generalization? After all, the total *number* of weights is unchanged, the network is simply encouraged to seek solutions that have *smaller* weights. How do solutions with smaller weights provide better generalization? The answer is that *not all weights are smaller*. Some weights will remain at relatively large values while others (that would normally take on larger values) will be forced to take on values near zero. The result is that the *average* weight size is smaller.

More specifically, the weights of the network fall into two categories: those that have a large influence on the solution, and those that have little of no influence on the solution. Let us refer to the weights in the second category as *excess weights*. Excess weights can take on a wide range of values without significantly affecting the solution. They are not likely to take on values near zero unless they are encouraged to do so. They are more likely to take on values that are either completely arbitrary, or that cause the network to overfit the data in order to gain a slight reduction in the training error. In either case the result is poor generalization. The addition of the second term in Eq. 24 to the criterion function encourages the excess weights to take on zero (or near zero) values. This improves generalization by discouraging overfitting.

When the weight decay method is used, it is easy to show that, once learning is complete, the magnitude of each weight in the network is directly proportional to its influence on the mapping error (the first term in Eq. 24). Thus, if we actually wish to delete weights from the network, we can simply delete the smallest weights produced by the weight decay process.

Weight decay is not without its drawbacks. It has the undesirable effect of biasing the weights that *are* influential in the solution toward slightly smaller values. This, in turn, biases the solution away from the one that minimizes the true mapping error [93].

An alternative technique which is based on the same approach is called *weight elimination* [142, 143]. In weight elimination the complexity regularization term is of the form:

$$J(w) = \sum_{p=1}^{P} J_p(w) + \lambda\sum_i \frac{w_i^2/w_o^2}{1+(w_i^2/w_o^2)} \tag{25}$$

where w_o is a fixed weight normalization factor. When $w_i > w_o$, the expression inside the sum is close to unity and this criterion essentially counts the number of weights. When $w_i < w_o$, the expression inside the sum is proportional to w_i^2 and this criterion works like the weight decay criterion. Through the appropriate choice of w_0 we can encourage the the network to seek solutions with a few large weights (w_0 small), or many small weights (w_0 large).

More recently a technique referred to as *soft weight sharing* has been proposed[102, 103]. This technique adds a term to the criterion function that encourages the network to use both zero and nonzero weights. In addition, it encourages the nonzero weights to cluster into different groups, so that weights from the same group have approximately the same value. This can be viewed as a type of weight sharing. In short, this technique combines the advantages of weight sharing and weight decay into a single unified approach.

Statistical Pattern Recognition

In statistical pattern recognition, the *optimal classifier* assigns the pattern **x** to a class ω according to *Bayes decision rule*. The two-class Bayes decision rule is given by:

$$P(\omega_1|x) \underset{\omega_2}{\overset{\omega_1}{\gtrless}} P(\omega_2|x) \tag{26}$$

The reader familiar with statistical decision theory will recognize this as the minimum risk decision rule with a zero-one loss function.

Equation 26 says: If $P(\omega_1|x)$ is greater than $P(\omega_2|x)$, then assign **x** to ω_1, otherwise assign **x** to ω_2. $P(\omega_i|x)$ is the *a posteriori* probability, and represents the probability that pattern **x** was drawn from class ω_i. Simply put, the decision rule says: "assign the pattern **x** to the class that it most probably belongs to." This decision rule is optimal in the sense that it minimizes the classification error, i.e., the average number of misclassifications.

Bayes decision rule can be (and often is) expressed in many different forms. In fact, any pair of functions $\{\varphi_1(x),\varphi_2(x)\}$ can be used in place of the *a posteriori* probabilities in Eq. 26 as long as they yield an equivalent decision rule. That is, the decision rule in Eq. 26 can be expressed as:

$$\varphi_1(x) \underset{\omega_2}{\overset{\omega_1}{\gtrless}} \varphi_2(x) \tag{27}$$

as long as:

$$\varphi_1(x) > \varphi_2(x) \quad \text{when} \quad P(\omega_1|x) > P(\omega_2|x) \tag{28}$$

and

$$\varphi_1(\boldsymbol{x}) < \varphi_2(\boldsymbol{x}) \quad \text{when} \quad P(\omega_1 | \boldsymbol{x}) < P(\omega_2 | \boldsymbol{x}) \tag{29}$$

Let us refer to φ_1 and φ_2 as *the simplified decision functions*. These decision functions can be obtained in a number of ways. If a functional description of the *a posteriori* probabilities is available then these decision functions can be obtained through simple algebraic manipulations of Eq. 26. This is a useful approach because it allows us to consider alternative implementations of the decision rule which may be easier to implement or computationally more efficient. A common example of this is the following two-class problem: both classes are equally likely, and their data have Gaussian distributions with equal covariance matrices and different mean vectors. Through a series of simple algebraic manipulations the decision functions can be reduced to linear classifiers [28, 134].

Although there is often an advantage to seeking simplified decision functions, there are some applications where the *a posteriori* probabilities are essential. These are applications where it is required that the data not only be assigned to the most probable class, but also that we know what that probability is. For example, suppose we are designing a system that classifies military targets as friend or foe. It may be essential that we know the probability that our decisions are correct, i.e., that we have some measure of confidence in our decisions. We are likely to react differently if our confidence is 99 percent as opposed to only 51 percent, even though the classification would be the same in both cases. The *a posteriori* probabilities provide these confidence estimates directly.

Although Bayes decision rule is quite simple, it is difficult to apply in practice because the *a posteriori* probabilities (and their corresponding family of decision functions) are usually unknown. This means that they must be estimated. There are numerous ways of estimating these functions. It is possible, for example, to use Bayes Rule to rewrite Eq. 26 in terms of *a priori* probabilities and density functions, i.e., $P(\omega_i | \boldsymbol{x}) = P(\omega_i) p(\boldsymbol{x} | \omega_i) / p(\boldsymbol{x})$. These probabilities and density functions can then be estimated using a variety of techniques described in the conventional pattern recognition literature [28, 134].

It is also possible to estimate the a posteriori probabilities directly. This problem is essentially one of functional approximation. Given a set of examples, we wish to train an estimator $\theta_i(\boldsymbol{x},\boldsymbol{w})$ to approximate $P(\omega_i | \boldsymbol{x})$. The vector **w** represents parameters of the estimator to be determined by the training procedure, e.g., weights in the MLP. Ideally we would expect the training set to be composed of example pairs of the form $(\boldsymbol{x}, P(\omega_i | \boldsymbol{x}))$. In practice however, we don't have access to the probability function, only the class labels ω_i. All is not lost, however, since it can be shown that when a mean-squared-error criterion is used for training, and 0s and 1s are used as the target outputs (in place of $P(\omega_1 | \boldsymbol{x})$ and $P(\omega_2 | \boldsymbol{x})$, respectively), the optimal solution for the parameters of $\theta_i(\boldsymbol{x},\boldsymbol{w})$ is the same as if the true *a posteriori* probabilities were used as the target outputs [28, 146]. Similar results exist for other types of criterion functions as well [7, 29, 40, 114]. It should be noted that this result is only true when the target outputs are 0s and 1s. Other target values, such as 0.1 and 0.9, can lead to biased results [84]. Thus, when we train with the Backpropagation algorithm, the MLP can learn the best mean-squared-error approximation to the *a posteriori* probabilities! This holds regardless of the estimator $\theta_i(\boldsymbol{x},\boldsymbol{w})$. Of course, this approximation can be very poor unless the estimator is capable of forming a good model of $P(\omega_i | \boldsymbol{x})$.

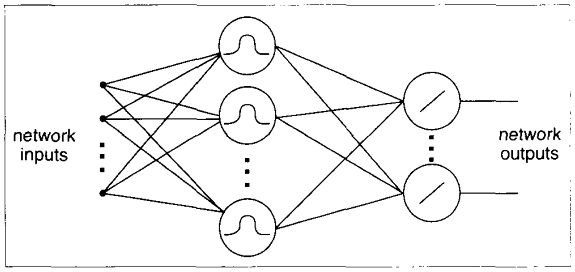
In general, choosing functions $\theta_i(\boldsymbol{x},\boldsymbol{w})$ that can serve as good estimators of $P(\omega_i | \boldsymbol{x})$ can be very difficult. These functions should have outputs that are bounded between zero and one and, in the multi-class case (3 or more classes), their outputs should all sum to unity. In the two-class case we only need to estimate one probability since the second is given by default, that is $P(\omega_2 | \boldsymbol{x}) = 1 - P(\omega_1 | \boldsymbol{x})$. These restrictions can severely limit the class of functions that we can consider, and can also complicate the learning procedure considerably. For this reason, it is more common to choose functions $\theta_i(\boldsymbol{x},\boldsymbol{w})$ that are good estimators of the simplified decision functions $\varphi_i(\boldsymbol{x})$. Estimators for $\varphi_i(\boldsymbol{x})$ are not nearly as restricted as those for $P(\omega_i | \boldsymbol{x})$. The parameters of $\theta_i(\boldsymbol{x},\boldsymbol{w})$ are determined in the same way as before, by using 0s and 1s as target outputs as if we were trying to estimate $P(\omega_i | \boldsymbol{x})$. The result is that we obtain good estimates of the decision functions $\varphi_i(\boldsymbol{x})$, but typically very poor estimates of the *a posteriori* probabilities $P(\omega_i | \boldsymbol{x})$. As a result, most systems are typically unable to provide confidence estimates with the class assignments. MLPs, on the other hand, can be very good estimators of the aforementioned function. Their sigmoid functions guarantee that the outputs are bounded between 0 and 1. In addition, it is not difficult to train these networks so that their outputs sum to one in the multi-class case [17, 27]. In fact, some authors have shown that the outputs tend to sum to a value close to one without applying any special constraints [14, 114]. Of course, the network must be the right size before the MLP can form a good approximation to $P(\omega_i | \boldsymbol{x})$. In addition, the number of training samples must also be sufficiently large to guarantee good generalization.

Accurate estimates of *a posteriori* probabilities are particularly important in applications where multiple estimates are combined in a higher level decision making process. One such example is in hidden Markov models (HMM) where MLPs have been used to estimate the emission and transition probabilities of the model [15]. One of the consequences of the fact that the network learns *a posteriori probabilities* is that the *a priori* probabilities, i.e., $P(\omega_i)$, can easily be adjusted after training. These probabilities, $P(\omega_i)$, represent the likelihood that a pattern will be drawn from class ω_i, prior to the actual presentation of the pattern. During learning, these probabilities are implicitly assumed to be equal to the relative number of patterns from each of the different classes in the training set. If the training set distribution does not accurately reflect the actual a priori probabilities, the network outputs can be scaled to compensate. This is possible because of the scalar relationship between the *a posteriori* probability estimate produced by the network and the *a priori* probability, i.e., $P(\omega_i | x) = P(\omega_i)\, p(x | \omega_i) / p(x)$. The proper adjustment can be made by scaling the estimate of $P(\omega_i | x)$ by $P(\omega_i)/P_t(\omega_i)$, where $P(\omega_i)$ is the true *a priori* probability, and $P_t(\omega_i)$ is the *a priori* probability implied by the training set distribution. Estimates of true *a priori* probabilities are easily obtained in most problems. In the zip-code recognition problem for example, the relative frequency of occurrence of the different digits can easily be determined from the large databases of zip codes that are available.

In summary, MLPs are good at both classification and in estimating *a posteriori* probabilities. In some applications this gives them a distinct advantage over other techniques.

Radial Basis Function Network

A Radial Basis Function (RBF) network [113] (Fig. 10) is a two-layer network whose output nodes form a linear combination of the basis (or kernel) functions computed by the hidden layer nodes. The basis functions in the hidden layer produce a localized response to input stimulus. That is, they produce a significant nonzero response only when the input falls within a small localized region of the input space. For this reason this network is sometimes referred to as the *localized receptive field* network [91, 92].



10. The radial basis function network.

Although implementations vary, the most common basis is a Gaussian kernel function of the form:

$$u_{1,j} = \exp\left[-\frac{(x - w_{1,j})^T (x - w_{1,j})}{2\sigma_j^2} \right] \quad j = 1, 2, \ldots, N_1 \tag{30}$$

where $u_{i,j}$ is the output of the jth node in the first layer, $\mathbf{x}$ is the input pattern, $w_{1,j}$ is the weight vector for the jth node in the first layer, i.e., the center of the Gaussian for node j; σ_j^2 is the normalization parameter for the jth node, and N_1 is the number of nodes in the first layer. The node outputs are in the range from zero to one so that the closer the input is to the center of the Gaussian, the larger the response of the node. The name "Radial Basis Function" comes from the fact that these Gaussian kernels are radially symmetric; that is, each node produces an identical output for inputs that lie a fixed radial distance from the center of the kernel $w_{1,j}$.

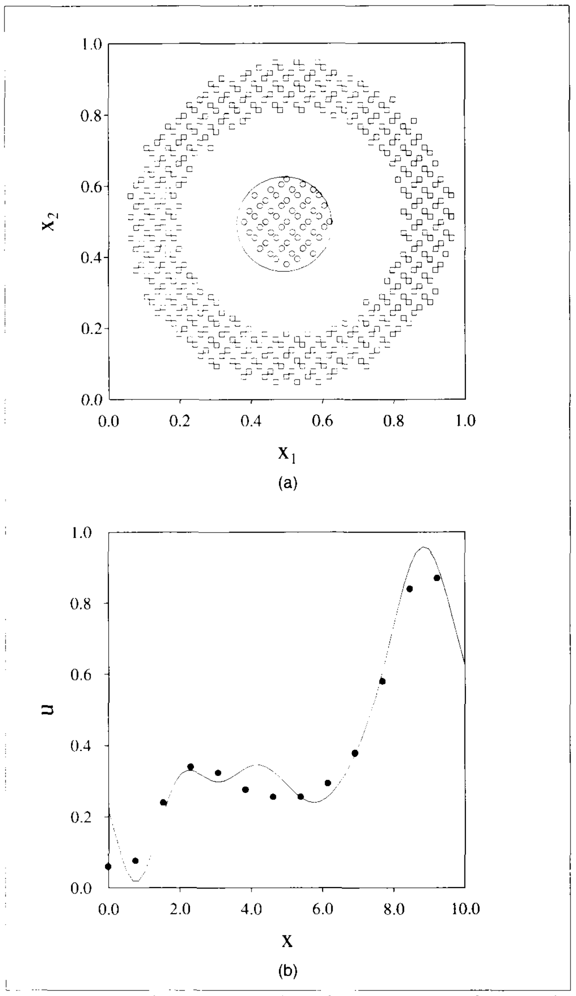
The output layer node equations are given by:

$$y_j = w_{2,j}^T \boldsymbol{u}_1 \quad j = 1, 2, \ldots, N_2 \tag{31}$$

where y_j is the output of the jth node, $w_{2,j}$ is the weight vector for this node, and $\boldsymbol{u}_1$ is the vector of outputs from the first layer (augmented with an additional component which assumes a value of one, just as we did in the perceptron). In addition, N_2 is the number of nodes in the output layer. The output layer nodes form a weighted linear combination of the outputs from the first layer. Thus, the overall network performs a nonlinear transformation from $\Re^N$ to $\Re^{N_2}$ by forming a linear combination of the nonlinear basis functions in Eq. 30.

Functional Capabilities of the RBF

The RBF network can be used for both classification and functional approximation, just like the MLP (Fig. 11). In the



11. Radial basis function examples. Classification (a); functional approximation (b).

```
procedure K_MEANS
  Initialize the cluster centers $\mathbf{w}_j, j = 1,2,\ldots,N_1$ ;
  /* typically these are set equal to the first $N_1$ training samples */
  repeat
    /* Group all patterns with the closest cluster center. */
    for all $\mathbf{x}_i$ do
      Assign $\mathbf{x}_i$ to $\Theta_{j^*}$, where $\mathbf{w}_{j^*} = \min_j \|\mathbf{x}_i - \mathbf{w}_j\|$ ;
    endloop
    /* Compute the sample means. */
    for all $\mathbf{w}_j$ do
      $\mathbf{w}_j = \frac{1}{m_j}\sum_{x_i \in j} x_i$ ;
    endloop
  until there is no change in cluster assignments from one iteration to the next ;
end ; {K_MEANS}
```

Table 3: K-Means Clustering Algorithm

classification example, only a single hidden layer node is required. The network effectively positions the Gaussian kernel at the center of the data, and then weights and thresholds it appropriately to produce the circular decision boundary shown. In the functional approximation example, the RBF network contains five hidden layer nodes. It is apparent from the figure that the Gaussian function centered 1.0 receives a negative weighting from the output layer while the other four Gaussians receive a positive weighting. It is instructive for the reader to compare these results with those in Fig. 5 for the MLP network.

In theory, the RBF network, like the MLP, is capable of forming an arbitrarily close approximation to any continuous nonlinear mapping [39, 46, 78, 111]. The primary difference between the two is in the nature of their basis functions. The hidden layer nodes in a MLP form sigmoidal basis functions which are nonzero over an infinitely large region of the input space, while the basis functions in the RBF network cover only small localized regions. While some problems can be solved more efficiently with sigmoidal basis functions, others are more amenable to localized basis functions. For example, in the classification problem in Fig. 11a, the RBF network provides a more efficient solution than the MLP network. The efficiency of the RBF network becomes even more pronounced as this problem is extended to higher dimensions (in higher dimensions, one class forms a complete shell around the other).

Regardless of the dimension, the RBF network will require only a single hidden layer node to solve this problem, but the MLP network will require on the order of n (the input dimension) hidden layer nodes. The reverse is true for the problem in Fig. 11b, where MLP network is more efficient.

RBF Learning Algorithms

There are a variety of approaches to learning in the RBF network. Most of them start by breaking the problem into two stages: learning in the hidden layer, followed by learning in the output layer [91, 92]. Learning in the hidden layer is typically performed using an unsupervised method, i.e., a clustering algorithm, while learning in the output layer is supervised. Once an initial solution is found using this approach, a supervised learning algorithm is sometimes applied to both layers simultaneously to fine-tune the parameters of the network.

There are numerous clustering algorithms that can be used in the hidden layer. A popular choice is the K-means algorithm shown in Table 3 [28, 134]. This algorithm is perhaps the most widely known clustering algorithm because of its simplicity and its ability to produce good results.

The normalization parameters, σ_j^2, are obtained once the clustering algorithm is complete. They represent a measure of the spread of the data associated with each node. Although they can be determined in a variety of ways the most common is to make them equal to the average distance between the cluster centers and the training patterns, that is:

$$\sigma_j^2 = \frac{1}{M_j}\sum_{x \in \Theta_j} (\boldsymbol{x} - \boldsymbol{w}_{1,j})^T (\boldsymbol{x} - \boldsymbol{w}_{1,j}) \tag{32}$$

where Θ_j is the set of training patterns grouped with cluster center $\boldsymbol{w}_{1,j}$, and M_j is the number of patterns in Θ_j. Learning in the output layer is performed after the parameters of the basis functions have been determined; that is, after learning in the first layer is complete. The output layer is typically trained using the Least Mean Squares (LMS) algorithm [149]. The training set consists of input/output pairs $(u_1, \boldsymbol{d})$ as before, but now the input patterns are processed by the first layer before being presented to the second layer for use in the training algorithm. The LMS algorithm is summarized in Table 4.

RBF Extensions

Extensions of the RBF network include variations on the basis functions, the learning algorithm, or both. A common variation on the basis functions is to increase their functionality (in an attempt to decrease the number of hidden nodes) by using the *Mahalanobis distance* in the Gaussian kernel [78, 97]. The basis function in Eq. 30 becomes

$$u_j = \exp\left[-(\boldsymbol{x} - \boldsymbol{w}_{1,j})^T \sum_j^{-1} (\boldsymbol{x} - \boldsymbol{w}_{1,j})\right] \quad j = 1, 2, \ldots, N_1 \tag{33}$$

where $\sum_j^{-1}$ is the normalization matrix for node j. These basis functions are no longer radially symmetric.

There are also numerous variations on learning. For example, once the initial phase of learning is completed as described above, it is sometimes useful to apply supervised

```
Procedure LMS
  Initialize the weights, $\mathbf{w}_{2,j}$, to small random value $j = 1,2, \ldots, N_2$ ;
  repeat
    Choose next training pair $(\mathbf{u}_1,\mathbf{d})$ ;
    /* Compute Outputs */
    for all $j$ do
      $y_j = \mathbf{w}_{2,j}^T \mathbf{u}_1$ ;
    endloop
    /* Compute Errors */
    for all $j$ do
      $e_j = y_j - d_j$ ;
    endloop
    /* Update Weights */
    for all $j$ do
      $\mathbf{w}_j(k+1) = \mathbf{w}_j(k) - \mu e_j \mathbf{u}_1$ ;
    endloop
  until termination condition reached ;
end ; {LMS}
```

Table 4: LMS Algorithm

learning to both layers simultaneously in an attempt to fine-tune the parameters in the hidden layer [19, 145].

Other variations on learning include techniques that select the centers of the Gaussians as a subset of the training samples [20]. In this method the samples are chosen one at a time in such a way that each new sample maximizes the amount of incremental gain in explaining the variance of the desired output. An orthogonal least squares algorithm is used to determine the output layer mapping. An Akaike-type criterion, which includes both a measure of "model fit" as well as "model complexity," is used to determine the number of hidden layer nodes. Nodes are no longer added when the incremental gain in model fit becomes smaller than the increase in model complexity. This algorithm is attractive because it includes a means for automatically determining the number of hidden layer nodes.

Other learning algorithms which incorporate methods for determining the number of hidden layer nodes can be found in [78, 97]. These methods focus primarily on pattern classification problems. During learning, a class membership is associated with each hidden layer node, i.e., each basis function. The number of hidden layer nodes is determined by using a type of supervised hierarchical clustering algorithm which either starts with one node and creates additional nodes as needed [78], or starts with a large number of nodes and merges them together whenever possible [97]. Both of these approaches also adapt the width of the basis functions during learning in an attempt to minimize the overlap between neighboring nodes of opposite classes.

Complexity of Learning in the RBF

One of the major advantages of the RBF network is that learning tends to be much faster than in the MLP [92]. The main reason for this is that the learning process is broken into two stages, and the algorithms used in both stages can be made relatively efficient. The first stage is intrinsically the most difficult since finding the optimal K-clustering for a set of data is NP-complete [37]. K-means is a greedy algorithm that finds a locally optimal solution, but generally produces good results and is usually very efficient. Other clustering algorithms, including some that are guaranteed to run in polynomial time, can be found in [28, 36, 45, 64, 134]. Once the hidden layer parameters are fixed, learning in the output layer is intrinsically easier. Because the network output is linear in the weights, the learning problem can be made polynomial. Although we have suggested the use of the LMS algorithm above, any learning algorithm for linear mappings could be used [28, 49, 134, 147].

It is not always the case that an optimal solution to the clustering problem yields an optimal set of hidden layer parameters for the RBF network. Thus, there are good reasons to apply a supervised learning algorithm to the hidden layer parameters. Several approaches were discussed above [19, 20, 78, 97, 145]. These algorithms can all be viewed as heuristic methods for finding good local solutions, but none are guaranteed to find the globally optimal solution. Although they tend to be more complicated than the simple two-stage approach that we presented, they offer efficient methods that can improve performance.

Generalization in the RBF

The basic issues related to generalization are the same for the RBF network as they were for the MLP network. We are concerned primarily with the number of training samples required for good generalization. As such, we are interested in determining the VCdim of the RBF network.

To get a handle on the upper bound, we can use the results of Baum and Haussler [10]. These results can be applied to any feedforward network with binary node outputs. Thus, assuming the outputs of the hidden layer nodes are thresholded to produce binary values, the VCdim of the RBF network can be shown [10, 70] to be bounded by:

$$VCdim \leq 2N_W \log(eN_N) \tag{34}$$

where N_W is the number of weights in the network, and N_N is the total number of nodes in the network. Of course the hidden layer nodes in the RBF network do not produce binary outputs, so this result must be used cautiously. In fact, much of the power of the RBF approach is due to the continuity of the basis functions. This bound is likely to be larger for nodes with continuous-valued outputs.

When the parameters in the hidden layer nodes are found in an unsupervised manner, the supervised learning procedure at the output layer cannot fully exploit the complete set of functions that the network is capable of implementing. Thus, to obtain a lower bound on the VCdim, we can simplify the generalization issue by treating the hidden layer as a preprocessing step, and focusing only on generalization as it relates to the supervised training at the output layer. The output layer consists of linear nodes, each with inputs of dimension N_1. The VCdim for one of these nodes is N_1+1, as we saw previously. This value then represents a lower bound on the VCdim of the RBF network.

Other Pattern Classifiers

Thus far, this paper has focused primarily on the Multilayer Perceptron and the Radial Basis Function networks, and their extensions. While these are not the only neural network pattern classifiers, we feel that they are arguably the most popular and most clearly illustrate the major features of neural network approaches to pattern classification. In this section, we briefly discuss some other pattern classifiers. These include more traditional techniques such as the Gaussian classifier [28, 36], Gaussian mixture methods [28, 36, 45], Parzen windows [28, 36], polynomial classifiers [28, 36], nearest-neighbor techniques [36], and tree-based methods [16]. There are as well other neural network approaches, including the Cerebellar Model Arithmetic Computer (CMAC) [3], the Probabilistic Neural Network (PNN) [129], and Learning Vector Quantization (LVQ) [67]. It is not our intent to develop each of these methods in detail, but rather to describe their basic modes of operation so that the reader can gain an appreciation for the relationships between them. By understanding how these different techniques relate to one another it can help to determine when neural network classifiers are appropriate.

The Gaussian classifier is a consequence of applying Bayes decision rule for the case where the probability functions for each class are assumed to be Gaussian. To implement this classifier, one need only form estimates of the mean vector and covariance matrix for each class of data, and substitute these estimates into the decision rule. The decision rule can be simplified to yield a polynomial classifier which is in general quadratic, but reduces to linear in the case where the covariance matrices for both classes are equal [28].

There are numerous ways of implementing the Gaussian classifier. The following three approaches are important because they form the foundation for the other classification schemes discussed in this section. The direct approach uses estimates of the class density functions which are substituted directly (without simplification) into Bayes rule. The discriminant function approach uses a simplified decision function which, as mentioned above, is a polynomial which is at most quadratic. The distance classifier approach uses a decision rule which assigns input patterns to the "closest class". In the Gaussian case, the closest class is determined by the distance from the input pattern to the class means. In the Gaussian problem all of these methods yield equivalent results, and all are optimal. But when these approaches are extended and applied to other problems their results can be quite different.

These three basic approaches can be extended by:

1. Using more elaborate density function estimates,
2. Using more powerful discriminant functions, or
3. Using a more general distance classifier.

More elaborate density estimates can be formed using a *Gaussian mixture* [28, 36, 45]. In this case the density function for each pattern class is approximated as a mixture of Gaussian density functions. Determining the optimal number of Gaussians in the mixture, and the parameters for each of the Gaussians can prove to be a difficult problem. Clustering algorithms like the K-Means algorithm can be used to determine the mean vectors of these distributions. Covariance matrices can then be estimated for each cluster once the mean vectors are determined. Such an approach should remind the reader of the Radial Basis Function network. The RBF network can be viewed as a weighted Gaussian mixture where the weights of the Gaussians are determined by the supervised learning algorithm at the output layer. In theory, this approach is capable of forming arbitrarily complex decision boundaries for classification.

An alternative approach to probability density function estimation is to place a window function, e.g., a Gaussian window, at *every training sample*. This type of approach is usually referred to as a Parzen window method. While the positions of the windows are defined automatically, determining the window widths can be a difficult task [28, 36]. Many neural network models are related to this approach. These models include the probabilistic neural network (PNN) [129], which uses a Gaussian window, and the CMAC model which uses a rectangular window [3]. As with the RBF network above, these approaches are capable of forming arbitrarily complex decision boundaries for classification, but they tend to consume more resources, in terms of both storage and computation, than the RBF network.

The methods discussed in the previous two paragraphs are sometimes called *local* methods because their receptive fields (window functions) provide a significant nonzero response for only a localized portion of the input space. The training patterns determine the positions and response of these receptive fields. New inputs will generate a response that is similar to the response generated by the training data that they resemble. In fact, if the windows are rectangular and nonoverlapping, the response to a new input pattern will be exactly that of the training pattern to which it is closest (assuming the input actually falls within one of the windows). In essence, the system is working as a *look-up table*. Typically, however, the windows are overlapping so that several receptive fields contribute to each network response.

These methods are often referred to as *memory-based models* since they represent generalizations of methods that work by "memorizing" the response to the training data. The usefulness of these techniques is generally determined by the efficiency with which they can cover the relevant portions of the input space. An inherent advantage of these techniques is that they respond only to inputs that are in the same regions of the input space as the training data.

The second of the three basic approaches mentioned above is the *discriminant function* approach. Perhaps the simplest discriminant function is a linear discriminant. This gives rise to a decision boundary between the classes that is linear, i.e., a hyperplane.

Linear discriminants are important, not only because of their practicality, but also because they have been shown to be optimal for many problems, including a special case of the Gaussian problem mentioned above. The Perceptron is a simple neural network classifier which employs a linear discriminant. The MLP can be viewed as an extension of the discriminant function approach which is capable of forming arbitrarily complex decision boundaries.

Tree-based classifiers can also be viewed as a nonlinear discriminant function approach. In a tree based-approach, the classifier is constructed by a series of simple greedy splits of the data into subgroups. Each subgroup is then split recursively, so that the resulting classifier has a hierarchical binary tree structure.

Typically, each greedy split simply finds the best individual component along which to split the data, although the algorithm can apply a more general split such as a linear or polynomial discriminant function at each node in the tree. Classification assignments are made at the leaves of the tree. The resulting decision boundary is generally a piecewise linear boundary. Typically, several leaves correspond to the same pattern class. The algorithm for building the tree is beyond the scope of our discussion here, but the interested reader is referred to [16]. This approach is particularly useful in problems where the input patterns contain a mixture of symbolic and numerical data. It also provides a rule-based interpretation of the classification method (the decisions made at each node determine the rules). Many of the techniques for growing neural networks that were previously discussed share the characteristics of tree-based classifiers.

The third approach is the distance classifier. The simplest distance classifier approach corresponds to the case of the Gaussian classifier with equal covariance matrices. In this case, the Euclidean distance from the input data sample to the mean of each class is used to make the classification decision. When the covariance matrices for the different classes are unequal, then the Euclidean distance must be replaced with the Mahalanobis distance. The resulting decision boundary is quadratic.

Extensions of the distance classifier approach include the k-nearest neighbor (k-NN) and the Learning Vector Quantization (LVQ) methods. The k-NN approach computes the distance between the input pattern and a set of labeled patterns, keeping track of the k-set of closest patterns from the labeled set. The input is then assigned to the class with the most members in the k-set. The labeled pattern set is formed directly from the training data. In fact, in the standard k-NN classifier, *all* training data are used in the labeled set. However, in the interest of reducing the computational and storage requirements, algorithms have been devised to reduce the size of the labeled set [36, 44]. The decision boundaries formed by this method are piecewise linear.

The LVQ method works exactly like a 1-NN classifier except that the set of labeled patterns is formed differently. This set is typically obtained by clustering the training data (to reduce the number of labeled patterns), and then using a supervised learning algorithm to move the cluster centers into positions that reduce the classification error [67].

Determining which classification method works best in a given application usually involves some degree of trial and error. Generally speaking, most of the approaches mentioned above can be designed to yield near-optimal classification performance to most problems. The real difference between them lies in other areas such as their time complexity of learning, their computational and storage requirements, their robustness, their number of free parameters (which govern generalization and sample size issues), and their potential use as estimators of a posteriori probabilities [56, 79, 80, 101].

Methods for Predicting Generalization Performance

Unless the training set is very large, the performance of the network on the training data is not likely to be an accurate measure of its performance on future data. Thus, there is a need for reliable methods of predicting the generalization performance of the network.

The standard method for predicting generalization performance is called *cross-validation*. This method works by splitting the data into two sets, a *training set* and a *test set*. Learning is performed on the training set, and network performance is evaluated on the test set. To achieve statistically significant results it is generally necessary to perform several independent splits, and then average the results to obtain an overall estimate of performance. While cross-validation is a widely accepted method, it can be extremely time consuming in neural networks because of the lengthy learning times required for each of the splits.

An alternative technique which requires far fewer calculations is called *predicted squared error* (PSE). This technique relys on statistical analysis methods to derive an expression for the generalization performance of a system as a function of its performance on the training set, the number of free parameters in the system, and the training set size. In particular, let us assume that the true function to be estimated is $F(\boldsymbol{x})$, and that our training set consists of noisy samples of this function such that the variance of the noise is σ^2; that is, $\mathrm{E}[(d-F(\boldsymbol{x}))^2]=\sigma^2$, where $(\boldsymbol{x},d)$ are input/output training pairs, and $\mathrm{E}[\cdot]$ is the expected value operator. Then, for systems which which contain $F(\boldsymbol{x})$ in their set of implementable functions, and are linear in the parameters, the predicted mean-squared-error is given by [8, 85, 93, 94]:

$$PSE = MSE + \frac{2N_w}{P}\sigma^2 \tag{35}$$

where MSE is the mean-squared-error on the training set, N_w is the number of free parameters (weights), and P is the number of training samples. It has been argued that this equation also provides an unbiased estimate of the predicted mean-squared-error for systems that are nonlinear in the parameters, e.g., neural networks, provided their mappings are sufficiently smooth [8]. Before the above expression can be used to predict network performance we must have a method for estimating σ^2. A standard estimate for this term [93] is:

$$\hat{\sigma}^2 = \left(\frac{P}{P - N_W} \right) MSE \tag{36}$$

Recently, Moody has proposed a similar formula that is applicable to systems which are trained using a complexity regularization term [93, 94]. This formula is called the *generalized prediction error* (GPE), and is of the form:

$$GPE = MSE + \frac{2N_{eff}}{P} \sigma^2 \tag{37}$$

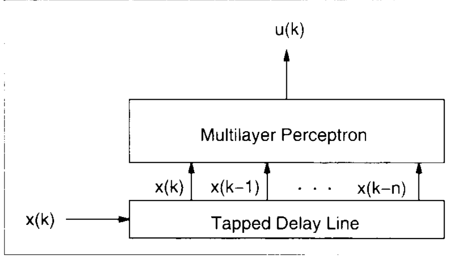
where N_{eff} is the *effective number of parameters* in the network. Because of complexity regularization, the effective number of parameters is typically much less than the actual number of parameters. Methods for estimating N_{eff} are discussed in [93, 94]. The corresponding estimate of the noise variance is the same as in Eq. 36, with N_W replaced by N_{eff}.

Dynamic Networks

The second class of networks we will discuss are dynamic networks. The node equations in these networks are described by differential or difference equations. These networks are important because many of the systems that we wish to model in the real world are nonlinear dynamical systems. This is true, for example, in the controls area in which we wish to model the forward or inverse dynamics of systems such as airplanes, rockets, spacecraft, and robots. Another class of nonlinear systems we wish to model are finite state machines which are at the heart of all modern day computers. Although these two examples may seem very different in nature, the networks discussed in this section can be used to model both. In this sense, they are very general models, with the potential for use in a wide variety of applications.

Time Delay Neural Network (TDNN)

Before we discuss networks that are truly dynamic, consider how an MLP is often used to process time series data. It is possible to use a static network to process time series data by simply converting the temporal sequence into a static pattern by unfolding the sequence over time. That is, time is treated as another dimension in the problem. From a practical point of view we can only afford to unfold the sequence over a finite period of time.



12. Time delay neural network.

This can be accomplished by feeding the input sequence into a tapped delay line of finite extent, then feeding the taps from the delay line into a static neural network architecture like a Multilayer Perceptron (Fig. 12). An architectures like this is often referred to as a Time Delay Neural Network (TDNN) [51]. It is capable of modeling systems where the output has a finite temporal dependence on the input, that is:

$$u(k) = F[x(k), x(k-1), ..., x(k-n)] \tag{38}$$

When the function $F(\cdot)$ is a weighted linear sum, this architecture is equivalent to a linear *finite impulse response* (FIR) filter. Because there is no feedback in this network, it can be trained using the standard Backpropagation algorithm.

The TDNN has been used quite successfully in many applications. The celebrated NETtalk project [124] used the TDNN for text-to-speech conversion. In this system, the input consisted of a local encoding of the alphabet and a small number of punctuation symbols. The output of the network was trained to give the appropriate articulary parameters to a commercial speech synthesizer. These signals represented the phoneme to be uttered at the point of text corresponding to the center character in the tapped-delay line. A version of the TDNN with weight sharing and local connections has been used for speech recognition with excellent results [73]. The TDNN has also been applied to nonlinear time series prediction problems [74]. The same approach using a Radial Basis Function Network in place of Multilayer Perceptrons was investigated in [92].

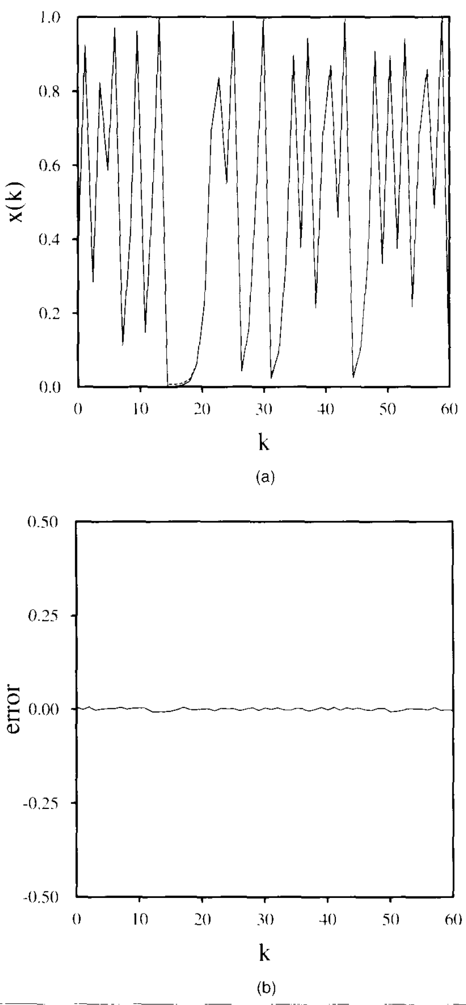
As an example, we trained the TDNN to perform 1-step prediction of the chaotic sequence:

$$x(k) = 4.0\, x(k-1)\, [1.0 - x(k-1)] \tag{39}$$

A two layer network was used with 2 hidden layer nodes. The input to the network was of order 1, i.e., only one delay was used in the tapped delay line. Figure 13 shows the actual and predicted sequences after training (the two are virtually indistinguishable).

Networks with Feedback Dynamics

Dynamical systems with feedback can offer great advantages over purely feedforward systems. For some problems, a small feedback system is equivalent to a large and possibly infinite feedforward system. For example, it is well known that an infinite number of feedforward logic gates are required to emulate an arbitrary finite state machine, or that an infinite order FIR filter is required to emulate a single pole infinite impulse response (IIR) filter [104, 130]. Systems with feedback are particularly appropriate for identification (modeling), control, and filtering applications. Most of the conventional work in these areas has been dominated by linear systems theory. But, as mentioned above, there are many problems which require nonlinear dynamics. Although the neural networks discussed here are by no means the only



13. *Prediction of the logistic equation with the TDNN. Chaotic sequence and its predicted value (a); error signal (b).*

approach to dynamical systems, they are interesting because of their neurological motivation and because of their applicability to a diverse set of nonlinear problems. This class of networks are commonly referred to as recurrent neural networks because they incorporate feedback and thus are inherently recursive.

One difficulty with recurrent networks is developing meaningful learning algorithms. For the most part, these learning algorithms are gradient search techniques similar to Backpropagation for Multilayer Perceptrons. Since the output of the nodes is a recursive function of the output of nodes on the previous time step, the calculation of the gradient must also be a recursive computation. This makes these learning algorithms considerably more complex.

Networks with Output Feedback

A simple way to incorporate feedback into a neural network architecture is to feed back the output of the network through a second tapped-delay line (Fig. 14). This particular architecture was introduced by Narendra [98] and has been used primarily for nonlinear identification and control problems [95]. This architecture is very general. In fact, if we apply the mapping theorems previously discussed for MLPs, then this architecture is capable, in theory, of modeling any system which can be expressed as:

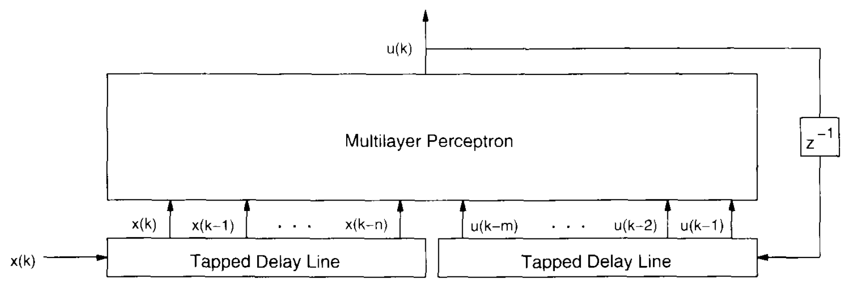
$$u(k)=F[x(k),x(k-1),\ldots x(k-n),u(k-1),u(k-2),\ldots,u(k-m)] \quad (40)$$

When the function $F(\cdot)$ is replaced by a weighted linear sum, this architecture is equivalent to an IIR filter. A constrained version of this architecture has also been proposed in which the outputs of the two tapped-delay lines are processed by two separate networks whose individual outputs are then summed to form the overall network output [57, 98]. The development of gradient descent learning algorithms for this architecture is beyond the scope of this article; however, the reader is referred to [57, 99] for more information.

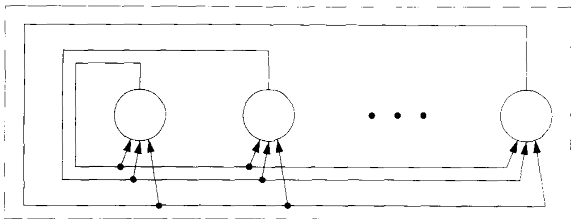
Networks with State Feedback

The next class of dynamic network models that we discuss incorporates a different type of feedback called state feedback. These networks are typically single-layer networks with feedback connections between nodes. In the most general case the nodes are completely interconnected, i.e., every node is connected to every other node, and also to itself (Fig. 15). Every node in the network contributes one component to the state vector. Any or all of the node outputs can be viewed as outputs of the network.

Additionally, any or all of these nodes may receive external inputs. This class of networks is perhaps the most general since many of the networks in previous sections can be obtained as simplifications of these.



14. *Narenda's dynamic neural network.*


15. *Recurrent network with state feedback.*

Continuous-Time Hopfield Net

The Hopfield network is probably the best known dynamic network model [53, 54]. It is a single-layer network with complete interconnections. The node equations for the continuous-time Hopfield network are given by:

$$\tau_i \dot{y}_i(t) = -y_i(t) + \sum_{j=1}^{N} w_{i,j} u_j(t) + v_i \qquad (41)$$
$$u_i(t) = f(y_i(t))$$

where $y_i(t)$ is the *internal state* of the ith neuron, $u_i(t)$ is the *output activation* (or *output state*) of the ith neuron, $w_{i,j}$ is the weight connecting the jth neuron to the ith neuron, and v_i is the input to the ith neuron.

The Hopfield network can be viewed as a nonlinear dynamical system with input vector $\boldsymbol{v}$, state vector $\boldsymbol{y}(t)$, and output vector $\boldsymbol{u}(t)$ as shown in Fig. 16. Because of the sigmoid nonlinearity, the output vector lies in the interior of an N-dimensional unit hypercube; that is, $\boldsymbol{u}(t) \in (0,1)^N$.

The Hopfield network is a nonlinear dynamical system which is capable of exhibiting a wide range of complex behavior. Depending on how the network parameters are chosen, it may function as a stable system, an oscillator, or even a chaotic system [2, 22, 32, 122]. Most of Hopfield's original applications required that the network perform as a stable system with multiple asymptotically stable equilibrium points. Conditions which guarantee this type of behavior are described below.

The asymptotic stability of a Hopfield network can be shown using Lyapunov's second method [137]. This method basically works by showing that the system is dissipating energy with time. This proof is accomplished by forming a Lyapunov function (or energy function) for the network, and showing that its time derivative is nonincreasing. The Lyapunov function for this network is defined as follows [23, 54]

$$E(t) = -\frac{1}{2}\boldsymbol{u}(t)^T \boldsymbol{W}\boldsymbol{u}(t) - \boldsymbol{u}(t)^T \boldsymbol{v} + \sum_{i=1}^{n} \int_{0}^{u_i(t)} f^{-1}(\alpha)\, d\alpha \qquad (42)$$

where $f(\cdot)$ is defined in Eq. 2. Note that Eq. 42 is not a typical Lyapunov function because it can take on negative values. However, it is easy to show that this function is bounded below, and can thus be made positive with an additive constant. This fact is sufficient for the proof of stability. Taking the time derivative of Eq. 42 yields:

$$\dot{E}(t) = \frac{dE(t)}{dt} = (\nabla_{\boldsymbol{u}} E(t))^T \dot{\boldsymbol{u}}(t) \qquad (43)$$

where ∇_u is the gradient operator with respect to **u**.

With symmetric **W**, we have:

$$\nabla_u E(t) = -\boldsymbol{W}\boldsymbol{u}(t) - \boldsymbol{v} + \boldsymbol{y}(t) \qquad (44)$$

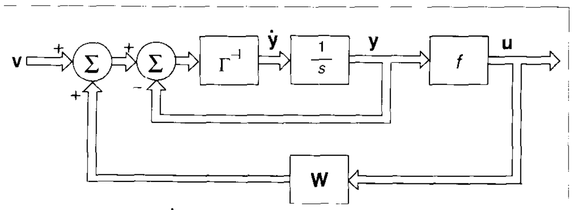
which from the equation in Fig. 16 is simply:

$$\nabla_u E(t) = -\Gamma \dot{\boldsymbol{y}}(t) \qquad (45)$$

With this, $\dot{E}(t)$ in Eq. 43 becomes:

$$\dot{E}(t) = -\dot{\boldsymbol{y}}(t)^T \Gamma \dot{\boldsymbol{u}}(t) \qquad (46)$$
$$= -\sum_{i=1}^{n} \tau_i \dot{y}_i(t)\, \dot{u}_i(t) = -\sum_{i=1}^{n} \tau_i \dot{y}_i(t)^2 \frac{\partial u_i}{\partial y_i}$$

Since τ_i and $\dot{y}_i(t)^2$ are always positive, we need only show that $\partial u_i / \partial y_i \geq 0$ to show that $\dot{E}(t) \leq 0$. This is trivially true, however, since $f(\cdot)$ in Eq. 2 is monotonically increasing. We also note from Eq. 46 that $\dot{E}(t)=0$ only when $\dot{\boldsymbol{y}}(t) = \boldsymbol{0}$. This completes the proof of asymptotic stability. Thus, under the condition that **W** is symmetric, the network will eventually reach a *fixed equilibrium point*. Furthermore, the locations of these equilibrium points, which can be found by setting $\dot{\boldsymbol{y}}(t)= 0$ in Fig. 16, are also extrema of E(t). This is easily verified from the above equations by noting that the extrema of $E(t)$ are defined by Eq. 45, with $\dot{\boldsymbol{y}}(t)= \boldsymbol{0}$.



$$\Gamma\dot{y}(t) = -y(t) + Wu(t) + v$$
$$u(t) = f(y(t))$$

16. *Continuous-time Hopfield network.*

The above result tells us that given any set of initial conditions $\boldsymbol{u}(0)$, the Hopfield network (with symmetric $\boldsymbol{W}$) will converge to a fixed equilibrium point, that is, to a point where $\dot{\boldsymbol{u}}(t)$=0. This equilibrium point, $\mathbf{u}_f$, is a fixed point in $(0,1)^n$. Because the network is deterministic, the location of this point is uniquely determined by the initial conditions. That is, the nonlinear nature of the Hopfield network gives rise to multiple equilibrium points, and the one chosen on any particular run of the network is determined uniquely by the initial conditions. All initial conditions that fall within the region of attraction of an equilibrium point will asympotically converge to that point. The exact number of equilibrium points, and their locations, are determined by the network parameters **W**, **v**, and β. At low gain (β small) the number of equilibrium points is small (possibly as small as 1), and their locations lie towards the interior of the hypercube. As the gain is increased, however, the number of equilibrium points grows large and their locations move towards the corners of the hypercube [54].

In the high-gain limit (β approaches infinity) the equilibrium points actually reach the corners of the hypercube and are maximum in number. This number is at most exponential in N [1, 87]. In addition, as β approaches infinity, one can show that the third term in Eq. 42 approaches zero, so that the energy function for the network simplifies to [54]:

$$E(t) = -\frac{1}{2}\boldsymbol{u}(t)^T \boldsymbol{W}\boldsymbol{u}(t) - \boldsymbol{u}(t)^T \boldsymbol{v} \tag{47}$$

The high-gain characteristics of the Hopfield network are of interest because many of the problems that we solve with this network require binary solutions; that is, we wish the equilibrium points, $\mathbf{u}_f$, to be binary vectors.

Once the gain is fixed (possibly at infinity so that hard-limiting nonlinearities are used), the locations of the equilibrium points are determined by **W** and **v**. When used to solve problems like the associative memory problem discussed below, the challenge is to design **W** and **v** so that the equilibrium points of the network correspond to solutions of the problem.

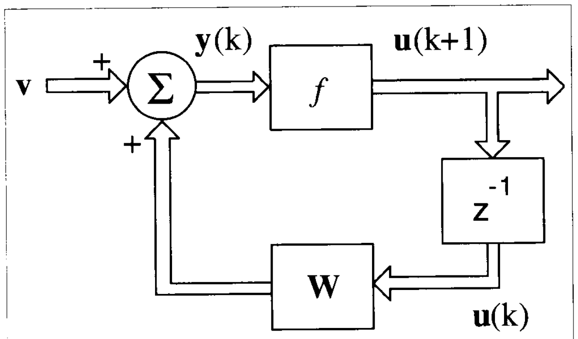
Discrete-Time Hopfield Network

A popular discrete-time version of the Hopfield network is described by the following node equations:

$$y_i(k) = \sum_{j=1}^{N} w_{i,j}\, u_j(k) + v_i \qquad u_i(k+1) = f_{HL}(y_i(k)) \tag{48}$$

where k is the time increment. These are a discrete-time approximation to Eq. 41, with the sigmoids replaced by hard-limiters. The behavior of this system is similar to Eq. 41: given any set of initial conditions $\mathbf{u}(0)$, and appropriate restrictions on the weights (given below), this network will converge to a fixed equilibrium point. Because a hard-limiting nonlinearity is used, these equilibrium points are binary vectors that are minima of Eq. 47. The system diagram for this network is shown in Fig. 17.

The stability and convergence properties of the discrete-time model are discussed in [18]. Sufficient conditions for stability are that **W** be symmetric and positive definite. Actually these conditions depend on the type of update that is used in the node equations. If *synchronous* updates are used, which implies that all nodes are updated simultaneously as suggested Fig. 17, then **W** must be positive definite; but when *asynchronous* updates are used (one node at a time) it is sufficient that the diagonal elements of **W** be nonnegative; that is $w_{ii} \geq 0$. These conditions are more stringent than those for the continuous-time network.



$$y(k) = Wu(k) + v$$
$$u(k+1) = f_{HL}(y(k))$$

17. Discrete-time Hopfield network.

Hopfield Associative Memory

One of Hopfield's original applications was the associative memory. An associative memory is a device which accepts an input pattern and produces as an output the *stored pattern* which is most closely associated with the input. In the Hopfield associative memory, the input/output patterns are binary. For example, the input pattern may be a noisy version of one of the stored patterns. The function of the associative memory is to recall the corresponding stored pattern, producing a clean version of the pattern at the output. In the Hopfield network the stored patterns are encoded in the weights of the network.

When used as an associative memory the input/output patterns are represented in bipolar form; that is, with 1s and -1s. Thus, the activation function in Eq. 1 is modified to yield a *bipolar* output.

The simplest approach for programming the weights of the Hopfield associative memory is to use the outer product method [53]. If we let z_i, i=1,2,...,M represent the M (bipolar) patterns that are to be stored in the network, then the weight matrix is programmed as follows:

$$\boldsymbol{W} = \sum_{i=1}^{M} z_i z_i^T - \alpha M \boldsymbol{I} \tag{49}$$

where $0 \leq \alpha \leq 1$. Note first that W is symmetric. Note also that each term in the summation contributes a value of +1 to the diagonal elements of W, so that the total contribution from the summation is M. The second term serves only to decrease the value of the diagonal elements by a fraction α. With α=0 the diagonal elements of W equal M, and with α=1 they equal 0. The effect of having zero verses nonzero diagonal elements in W is discussed in [18]. The storage and recall properties of this method are well known. It has been shown that if the input patterns are less than N/2 away in Hamming distance from a stored pattern, and if the M stored patterns are chosen at random, the maximum asymptotic value of M for which all M of the patterns can be perfectly recalled is [71, 87]:

$$M \leq \frac{N}{4 \ln N} \tag{50}$$

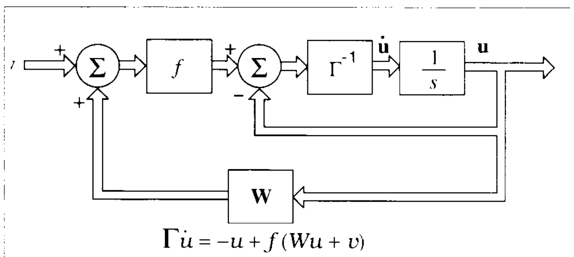
For example, if $N = 256$ then $M \leq 12$. Alternatively, if we allow a small fraction of the bits in the recalled pattern to be in error, then the capacity is $M \leq 0.138\,N$ [51]. With $M=0.138\,N$, approximately 1.6 percent of the bits in the recalled pattern are in error. For M greater than $0.138\,N$, the number of erroneous bits increases rapidly (an avalanche effect), rendering the network useless as an associative memory. These results assume that the weight matrix is formed using the outer product design method in Eq. 49. Other design techniques which provide improved storage capacity can be found in [31, 136]. The capacities achieved with these methods are closer to $M = N$, which is much better than those achieved with the outer product method, and are in fact the maximum possible for the Hopfield network [1].

Continuous-Time Recurrent Neural Network

The next model that we discuss is very similar to the Hopfield network. This model is called the Continuous-Time Recurrent Neural Network (CTRNN), and is described in [109]. Like the Hopfield network, this network consists of a single layer of nodes which are fully interconnected. Generally, one subset of the nodes serve as the "input nodes," while another serves as the "output nodes." These subsets are denoted A and Ω, respectively. The dynamics of the network are described by the following differential equation:

$$\tau_i\,\dot{u}_i(t) = -u_i(t) + f\Big(\sum_{j=1}^{N} w_{i,j}\,u_j(t) + v_i\Big) \quad i = 1,2,\ldots,N \tag{51}$$

where $u_i(t)$ is the state of the ith node, $w_{i,j}$ is the weight which connects node j to node i, v_i is the input to the ith node, and $f(\cdot)$ is the nonlinear activiation function (typically a sigmoid). The dynamics of this system are shown in Fig. 18. It is instructive to compare this diagram with that of the Hopfield network in Fig. 16; the two models are closely related. In fact, one can show that the Hopfield Network is related the CTRNN by a simple affine transformation. If we let $\boldsymbol{y}_H(t) = \boldsymbol{W}\boldsymbol{u}_R(t) + \boldsymbol{v}$, where $\boldsymbol{y}_H(t)$ is the state vector of the Hopfield network and $\boldsymbol{u}_R(t)$ is the state vector of the recurrent neural net, then the defining equations in Fig. 18 can easily be obtained from the equations accompanying Fig. 16. (This transformation requires that $\mathbf{W}$ be invertible.) Training algorithms for the CTRNN are discussed in [107, 109].



18. Continuous-time recurrent neural network.

Discrete-Time Recurrent Neural Network

A discrete-time approximation to the CTRNN is the discrete-time recurrent neural network (DTRNN) [117, 150]. Consider the following discrete-time approximation of Eq. 51:

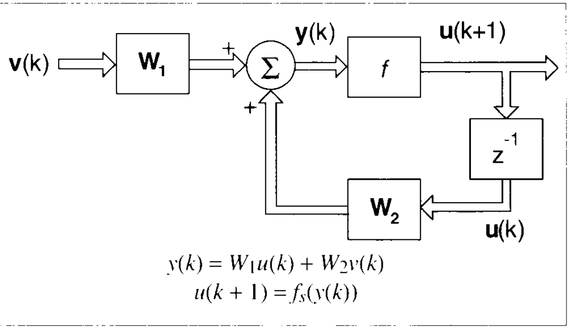
$$u_i(k+1) = f\Big(\sum_{j=1}^{N} w_{i,j}\,u_j(k)\Big) + v_i(k)) \tag{52}$$

Now if we allow each node to weight not only the outputs from other nodes, but also the components of the input vector, then we can re-express the above equation as:

$$u_i(k+1) = f\Big(\sum_{j=0}^{N+M} w_{i,j}\,u_j(k)\Big) \tag{53}$$

where $u_i(k)$ is reexpressed as:

$$u_i(k) = \begin{cases} 1 & i = 0 \\ u_i(k) & i = 1,2,\ldots,N \\ v_{i-N}(k) & i = N+1,\ldots,N+M \end{cases} \tag{54}$$



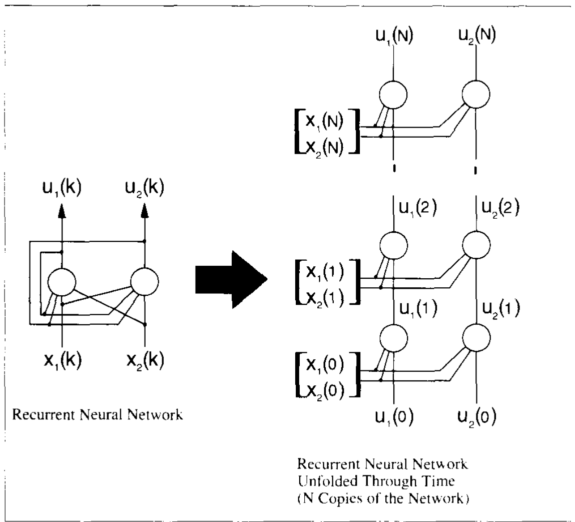
19. Discrete-time recurrent neural network.

N is the number of nodes in the network, and M the size of the input vector $\mathbf{v}$. We define $u_0(k)=1$ to account for the bias weights, $w_{i,0}$, just as we did in the MLP. Equation 53 then describes the node equations for the DTRNN. The system diagram for this network is shown in Fig. 19.

DTRNN Functional Capabilities

The behavior of the DTRNN is similar to the CTRNN discussed above. However, the DTRNN can take on a unique interpretation based on its discrete-time nature. Recall that each node in the network is capable of implementing simple logic functions. This property, along with the feedback connections and the discrete time delays, make it possible for the DTRNN to emulate Deterministic Finite Automata (DFAs) [4, 88]. With this interpretation, it is easy to see how the DTRNN can perform such tasks as sequence recognition.

It is also easy to show that the DTRNN can be reduced to an ordinary MLP. First we note that when the connection matrix $\boldsymbol{W}_2$ is lower triangular, all weight connections are feed



20. Unfolding a recurrent network into a static network with shared weights.

forward. Then with some additional constraints we can form a weight matrix which imposes a layered structure onto the nodes. The resulting system is identical to an MLP except that it takes n time steps to feed a pattern from the input to the output layer of an n layer network.

DTRNN Learning Algorithms

One way to learn the weights in a Discrete-Time Recurrent Neural Network is to convert the network from a feedback system into a purely feedforward system by *unfolding the network over time*. The idea is that if the system processes a signal that is n time steps long, then we create n copies of the network. The feedback connections are modified so that they are now feedforward connections from one network to the subsequent network (Fig. 20).

The network can then be trained as if it is one giant feedforward network with the copied weights being treated as shared weights. This approach to learning is called *Backpropagation Through Time* [119]. Another approach, called *Truncated Backpropagation Through Time*, tries to approximate the true gradient by only unfolding the network over the last m time steps [151]. In this case, only m copies of the network are made, and normal Backpropagation with weight sharing is used. For some problems performance may be sacrificed if critical information occurs more than m time steps in the past. The advantage of these learning algorithms is that the computation is simplified, since normal Backpropagation with weight sharing can be used. However, there is a large memory cost to maintain several copies of the network.

Another approach is to calculate the gradient recursively. This approach is commonly known as *Real Time Recurrent Learning* (RTRL) [150]. The advantage of this approach is that it is more memory efficient than Backpropagation Through Time, although there is a computational cost incurred by the recursive process. Here again, the learning algorithm is derived using a gradient search to minimize a Sum of Squared Error criterion. The criterion function is given by:

$$J(\boldsymbol{w}) = \sum_{p=1}^{P} J_p(\boldsymbol{w}) \qquad (55)$$

where P is the number of training sequences, and $J_p(\boldsymbol{w})$ is the total squared error for the pth sequence:

$$J_p(\boldsymbol{w}) = \frac{1}{2}\sum_{k=1}^{K_p} \sum_{j\in\Omega} (d_j(k) - u_j(k))^2 \qquad (56)$$

In this equation, K_p is the length of the pth training sequence. Recall also that Ω represents the set of output nodes in the network. Applying the gradient operator to Eq. 55 and substituting into the weight update formula yields:

$$w_{j,i}(m+1) = w_{j,i}(m) - \mu \sum_{p=1}^{P} \left.\frac{\partial J_p(\boldsymbol{w})}{\partial w_{j,i}}\right|_{\boldsymbol{w}(m)} \qquad (57)$$

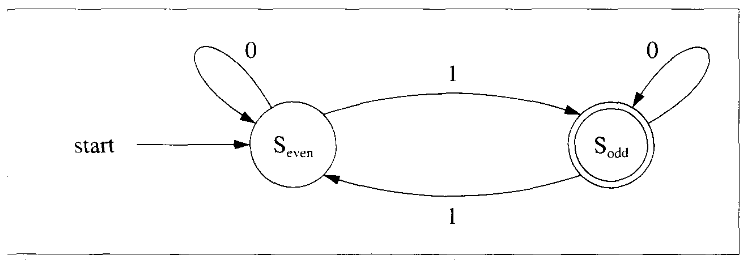
where m is the iteration index of the weights. Note that m is necessarily different from k. The later is an index on the time scale of the input sequences, while the former is an index on the time scale of weight updates which is much slower. The partial derivative in Eq. 57 is of the form [150]:

$$\frac{\partial J_p(\boldsymbol{w})}{\partial w_{j,i}} = -\sum_{k=1}^{K_p} \sum_{h\in\Omega} (d_h(k) - u_h(k)) p_{j,i}^{h}(k) \qquad (58)$$

where $p_{j,i}^{h}(k)$ is a partial derivative defined as:

$$p_{j,i}^{h}(k) = \frac{\partial u_h(k)}{\partial w_{j,i}} \qquad (59)$$

and can be expressed as:



21. DFA that recognizes arbitrary length strings with an odd number of 1's.

$$p_{j,i}^{h}(k) = f'\Big(\sum_{\alpha=0}^{N+M} w_{h,\alpha}\, u_\alpha(k-1)\Big) \cdot \Big[\delta_{h,j}\, u_i(k-1) + \sum_{\beta=1}^{N} w_{h,\beta} \frac{\partial u_\beta(k-1)}{\partial w_{j,i}}\Big]$$
$$= f'\Big(\sum_{\alpha=0}^{N+M} w_{h,\alpha}\, u_\alpha(k-1)\Big) \cdot \Big[\delta_{h,j}\, u_i(k-1) + \sum_{\beta=1}^{N} w_{h,\beta}\, p_{j,i}^{\beta}(k-1)\Big] \tag{60}$$

where $\delta_{h,j}$ is the Kronecker delta function. Note that in an n node network, we must maintain n terms for each weight. Since there are n^2 weights, there are n^3 of these terms. In addition, the calculation of each of these terms requires a summation over n terms (the sum over β). Thus, each time iteration requires $O(n^4)$ calculations.

Finally, substituting Eq. 58 into equation 57 gives:

$$w_{j,i}(m+1) = w_{j,i}(m) + \mu \sum_{p=1}^{P} \sum_{k=1}^{K_p} \sum_{h \in \Omega} (d_h(k) - u_h(k))\, p_{j,i}^{h}(k) \tag{61}$$

Note in this equation that the weights are updated only once each time through the training set. However, it may be desirable to perform updates each time a sequence is presented to the network.

Recall that in the derivation of the Backpropagation algorithm the true gradient was approximated by an instantaneous estimate based on a single sample. Using a similar approach here, one obtains the following approximation:

$$w_{j,i}(m+1) = w_{j,i}(m) - \mu \left. \frac{\partial J_{m \bmod P}(\mathbf{w})}{\partial w_{j,i}} \right|_{\mathbf{w}(m)} \tag{62}$$

The learning algorithm with this approximation is summarized in Table 5.

An example

Logic problems were among the first to be considered for learning in static networks because of their solid theoretical foundation. It seems natural then that Deterministic Finite Automata (DFAs) (logic gates coupled with a finite memory) be among the first problems investigated in recurrent neural networks. We have seen that Multilayer Perceptrons are capable of implementing arbitrary logic functions if we allow a sufficient number of hidden layer nodes. Similarly, it can be shown that the DTRNN is capable of implementing arbitrary DFAs [4]. Of course, we are interested primarily in *learning* DFAs from a set of examples.

Since the XOR problem has a great deal of historical significance, it is natural to look at its time-dependent counterpart— the *parity problem*. The DFA for parity is shown in Fig. 21. The circles in the figure correspond to the *states* of the DFA. There are two states, S_{even} and S_{odd}. The directed arcs between the states are *transitions* indicating how the states change as the input string is processed. We start in state S_{even} and process the input string one character at a time, making transitions between the states. For this DFA, the state labeled S_{even} corresponds to the case when the number of '1's in any prefix of the input string is an even number, while the state labeled S_{odd} corresponds to an odd number of '1's. The double circle on state S_{odd} indicates that if we end in this state after processing an input string, then that string is *accepted* by the DFA; otherwise the string is *rejected*. Thus, this DFA accepts all strings which have an odd number of 1's and rejects all other strings.

The problem of learning, or *inferring* DFAs is typically stated as follows: Given a set of positive and negative example strings, find a DFA which accepts the positive strings and rejects the negative strings. It turns out that such problems are relatively easy [35] unless we limit the number of states that the DFA can have, in which case the problem is NP-complete [6, 41].

A DTRNN with 3 nodes, one of which was chosen as the output node, was trained to solve the parity problem. The training set consisted of a set of 1000 strings randomly sampled from the set of all strings of length up to and including length five. Thus, many of these strings were presented repeatedly to the network. We found it advantageous to repeat the shorter strings more frequently than the longer strings. In our case each string appeared with a relative frequency of occurrence equal to:

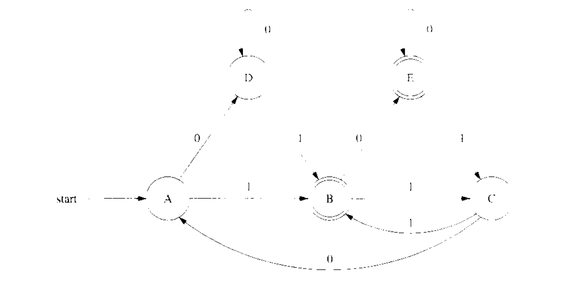
$$p_i = \frac{\left(\frac{1}{2}\right)^{l_i}}{\sum_j \left(\frac{1}{2}\right)^{l_j}} \tag{63}$$

where l_i is the length of the ith string. A popular alternative approach is to train on shorter strings first, and then add longer strings in the later stages of learning [82]. The network found a solution after 91 training epochs. To determine the DFA learned by the network, we replaced the sigmoids with hard limiting nonlinearities and extracted the logic functions implemented by the network. The resulting DFA is shown in Fig. 22. While this DFA looks considerably more complex than the DFA shown in Fig. 21, it can easily be shown that the two DFAs are equivalent.

The process of *extracting* the DFA from a recurrent neural network is not, in general, as easy as replacing sigmoids by hard limiters [38, 141]. Often the network utilizes the transition region of the sigmoid to encode the states of the DFA which can greatly complicate matters. Some researchers have found that the representations of the states in the DFA can be distributed throughout the state space in a chaotic manner,

```
procedure DTRNN
  Initialize the weights ;
  set $m = 1$ ;
  repeat
    Choose next training sequence $\{\mathbf{v}(k), \mathbf{d}(k)\}$, $k = 1, \ldots, K_p$ ;
    Set all states to zero ;
    Set all $g_{j,i}(m)$ to zero ;
    for $k = 1$ to $K_p$ do
      for all $u_i$ do
        $u_i(k) = f(\sum_{j=0}^{N+M} w_{i,j}(m) u_j(k-1)$;
      endloop
      for all $p_{j,i}^{h}$ do
        $p_{j,i}^{h}(k) = f'(\sum_{\alpha=0}^{N+M} w_{h,\alpha}(m) u_\alpha(k-1))[\delta_{h,j} u_i(k-1) + \sum_{\beta=1}^{N} w_{h,\beta}(m) p_{j,i}^{\beta}(k-1)]$ ;
      endloop
      for all $g_{j,i}$ do
        $g_{j,i}(m) = g_{j,i}(m) + \sum_{h \in \Omega}(d_h(k) - u_h(k)) p_{j,i}^{h}(k)$ ;
      endloop
    endloop
    for all $w_{j,i}$ do
      $w_{j,i}(m+1) = w_{j,i}(m) + \mu g_{j,i}(m)$ ;
    endloop
    Set $m = m + 1$ ;
  until termination condition reached ;
  end; {DTRNN}
```

Table 5: Real-Time Recurrent Learning with the DTRNN



22. DFA extracted from the DTRNN.

indicating that recurrent networks may be capable of implementing much more complex machinery than simple DFAs [112]. In fact, it has been shown that theoretically the DTRNN is Turing equivalent [125].

Summary

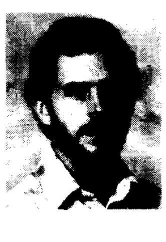
This article has summarized some of the recent developments in neural networks. Of course, space constraints make it impossible to cover all of the relevant work in this area, so we have restricted our focus to the network models that were probably the most theoretically immature at the time of Lippmann's 1987 paper, and recent extensions to these models. This includes the most popular networks to emerge in the 1980s such as the Multilayer Perceptron, the Hopfield network, and various Recurrent Network models. For additional material, the reader is referred to the numerous text books and edited volumes that have appeared in recent years, many of which are cited in the reference section.

In addition, neural networks are now becoming a part of many conferences and journals in engineering, computer science, cognitive science, and physics. Conferences which are devoted entirely to neural networks include *Neural Information Processing Systems—Natural and Synthetic*, (NIPS) sponsored by the IEEE Information Theory Group; *World Congress on Neural Networks*, (WCNN) sponsored by the International Neural Network Society; *IEEE International Conference on Neural Networks*, (IEEE-ICNN) sponsored by the IEEE Neural Networks Council; *International Conference on Artificial Neural Networks*, (ICANN) sponsored by the European Neural Network Society, and *IEEE Workshop on Neural Networks for Signal Processing* sponsored by the IEEE Signal Processing Society. Journals devoted entirely to neural networks include *IEEE Transactions on Neural Networks*, *Neural Networks*, *Neural Computation*, *Neurocomputing*, and *Network*.

Acknowledgment

We would like to acknowledge the contribution of Dr. Chaouki Abdallah and Dr. Greg Heileman for their help in critiquing the manuscript. We would thank the reviewers for their time and consideration. Their recommendations were very helpful. Thanks also to the BSP Group for their enlightening series of Friday afternoon seminars at the UNM Club. Finally, we would like to thank Sandia National Laboratories, Los Alamos National Laboratories, and the National Aeronatics and Space Administration for their support of our endeavors in this area.

Don R. Hush received the B.S.E.E. and M.S.E.E. degrees from Kansas State University, Manhattan, KS in 1980 and 1982, respectively, and his Ph.D. in engineering at the University of New Mexico, Albuquerque, in 1986. From 1986 to 1987 he worked at Sandia National Laboratories. He is currently an Assistant Professor in the Electrical and Computer En-



gineering Department at the University of New Mexico. He is a Senior Member of the IEEE, and a member of the International Neural Network Society. He is the coauthor of a 1990 Prentice-Hall text entitled Digital Signal Analysis. His research interests include neural networks, pattern recognition, signal processing, and controls.

Bill Horne received his B.S.E.E. from the University of Delaware, Newark, DE in 1986, and his M.S.E.E. at the University of New Mexico in 1988. He is currently pursuing his Ph.D. in Electrical Engineering at the University of New Mexico, where the topic of his dissertation is the circuit complexity of recurrent neural network implementations of finite state machines. His research interests involve neural networks, control theory and theoretical computer science.

References

[1] Y.S. Abu-Mostafa and J.M. St. Jaques, "Information capacity of the Hopfield model," IEEE Transactions on Information Theory, 31:461-464, 1985.

[2] K. Aihara, T. Takabe, and M. Toyoda, "Chaotic neural networks," *Physics Letters*, A, 144:333-340, 1990.

[3] J.S. Albus, "Mechanisms of planning and problem solving in the brain." *Mathematical Biosciences*, 45:247-293, 1979.

[4] N. Alon, A.K. Dewdney, and T.J. Ott, "Efficient simulation of finite automata by neural nets," *Journal of the Association of Computing Machinery*, 38(2):495-514, 1991.

[5] J.A. Anderson and E. Rosenfeld, *Neurocomputing*. MIT Press, Cambridge, MA, 1989.

[6] D. Angluin, "On the complexity of minimum inference of regular sets," *Information and Control*, 39:337-350, 1978.

[7] A.R. Barron, "Statistical properties of artificial neural networks," In *Proceedings of the 28th IEEE Conf. on Decision and Control*, pp. 280-285, 1989.

[8] A.R. Barron and R. Barron, "Statistical learning networks: A unifying view." In E.J. Wegman, D.I. Gantz, and J.J. Miller, editors, *Computing Science and Statistics: Proc. of the 2Oth Symposium on the Interface*, pp. 192-202, 1989.

[9] A.R. Barron, F.W. van Straten, and R.L. Barron, "Adaptive learning network approach to weather forcasting: a summary." In *Proceedings of the IEEE Int. Conf. on Cybernetics and Society*, pp. 724-727, 1977.

[10] E.B. Baum and D. Haussler, "What size net gives valid generalization?" *Neural Computation*, 1:151-160, 1989.

[11] S. Becker and Y. le Cun, "Improving the convergence of back-propagation learning with second order methods." In *Proceedings of the 1988 Connectionist Models Summer School*, pages 29-37. Morgan Kaufmann, 1988.

[12] S. Becker and Y. le Cun, "Improving the convergence of back-propagation learning with second-order methods," Technical Report CRG-TR-88-5, U. of Toronto, Toronto, Canada, 1988.

[13] A. Blum and R.L. Rivest, "Training a 3-node neural network is NP-complete," In *Proceedings of the Computational Learning Theory (COLT) Conference*, pp. 9-18. Morgan Kaufmann, 1988.

[14] H. Bourlard and N. Morgan, "A continuous speech recognition system embedding MLP into HMM," In D. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pp. 186-193. Morgan Kaufmann, 1990.

[15] H. Bourlard and C.J. Wellekens, "Links between markov models and multilayer perceptrons, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(12): 1167- 1178, 1990.

[16] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone, *Classification and Regression Trees*, Wadsworth & Brooks, Pacific Grove, CA, 1984.

[17] J.S. Bridle, "Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters," In D. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pp. 211-217. Morgan Kaufmann, 1990.

[18] J. Bruck and J.W. Goodman, "A generalized convergence theorem for neural networks," *IEEE Transactions on Information Theory*, 34:1089-1092, September 1988.

[19] S.V. Chakravarthy, J. Ghosh, L. Deuser, and S. Beck, "Efficient training procedures for adaptive kernel classifiers." In *Proceedings of the First IEEE-SP Workshop on Neural Networks for Signal Processing*, Princeton, NJ, pages 21-29, 1991.

[20] S. Chen, C.F.N. Cowan, and P.M. Grant, "Orthogonal least squares learning algorithm for radial basis function networks, *IEEE Transactions on Neural Networks*, 2(2):302-309, 1991.

[21] D.L. Chester, "Why two hidden layers are better than one." In *Proceedings of the International Joint Conference on Neural Networks*, volume 1, pp. 265-268, Erlbaum, 1990.

[22] M.Y. Choi and B.A. Huberman, "Dynamic behavior of nonlinear neural networks," *Physical Review*, A, 28:1204-1206, 1983.

[23] D.H. Cohen and S.M. Sherman, "The nervous system and its components," In R.M. Berne and M.N. Levy, editors, *Physiology*, pages 69-76, C.V. Mosby, St. Louis, MO, 1983.

[24] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals, and Systems*, 2(4):303-314, 1989.

[25] E.D. Dahl, "Accelerated learning using the generalized delta rule," In *Proceedings of the IEEE 1st International Conference on Neural Networks*, volume 2, pp. 523-530, San Diego, CA, 1987.

[26] J. Denker, D. Schwartz, B. Wittner, S. Solla, R. Howard, L. Jackel, and J. Hopfield, "Large automatic learning, rule extraction, and generalization," *Complex Systems*, 1:877- 922, 1987.

[27] J.S. Denker and Y. le Cun, "Transforming neural-net output levels to probability distributions," In R.P. Lippmann, J.E. Moody, and D. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pp. 853-859. Morgan Kaufmann, 1991.

[28] R.O. Duda and P.E. Hart, *Pattern Classification and Scene Analysis*. Wiley, New York, NY, 1973.

[29] A. El-Jaroudi and J. Makhoul, "A new error criterion for posterior probability estimation with neural nets," In *Proceedings of the International Joint Conference on Neural Networks*, volume 3, pages 185-192, 1990.

[30] S.E. Fahlman and C. Lebiere, "The cascade-correlation learning architecture." In D. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pp. 524-532. Morgan Kaufmann, 1990.

[31] J.A. Farrell and A.N. Michel, "A synthesis procedure for Hopfield's continuous-time associative memory," *IEEE Transactions on Circuits and Systems*, 37:877-884, July 1990.

[32] W.J. Freeman and Y. Yao, "Model of biological pattern recognition with spatially chaotic dynamics," *Neural Networks*, 3:153-170, 1990.

[33] J.H. Friedman and W. Stuetzle, "Projection pursuit regression," *J. Amer. Stat. Assoc.*, 76:817-823, 1981.

[34] J.H. Friedman and J.W. Tukey, "A projection pursuit algorithm for exploratory data analysis, IEEE Transactions on Computers, 23:881-889, 1974.

[35] K.S. Fu and T.L. Booth, "Grammatical inference: Introduction and survey— Part I," *IEEE Transactions on Systems, Man and Cybernetics*, 5:95-111, 1975.

[36] K. Fukunaga, *Introduction to Statistical Pattern Recognition*. Academic

Press, San Diego, CA, 1972.

[37] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, New York, NY, 1979.

[38] C.L. Giles, C.B. Miller, D. Chen, H.H. Chen, G.Z. Sun, and Y.C. Lee, "Learning and extracting finite state automata with second-order recurrent neural networks." *Neural Computation*, 4(3):393-405, 1992.

[39] F. Girosi and T. Poggio, "Networks and the best approximation property." Artificial Intelligence Lab. Memo 1164, MIT, 1989.

[40] H. Gish, "A probabilistic approach to the understanding and training of neural network classifiers." In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 1361-1364, April 1990.

[41] E.M. Gold, "Complexity of automaton identification from given data." *Information and Control*, 37:302-320, 1978.

[42] A. Hajnal, W. Maass, P. Pudlak, M. Szegedy, and G. Turan, "Threshold circuits of bounded depth." In *Proceedings of the 1987 IEEE Symposium on the Foundations of Computer Science*, pp. 99-110, 1987.

[43] S.J. Hanson and L.Y. Pratt, "Comparing biases for minimal network construction with back-propagation." In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, pp. 177-185. Morgan Kaufmann, 1989.

[44] P.E. Hart, "The condensed nearest neighbor rule." *IEEE Transactions on Information Theory*, 114:515-516, 1968.

[45] J.A. Hartigan. *Clustering Algorithms*. Wiley, New York, 1975.

[46] E.J. Hartman, J.D. Keeler, and J.M. Kowalski, "Layered neural networks with Gaussian hidden units as universal approximations." *Neural Computation*, 2(2):210-215, 1990.

[47] B. Hassibi and D.G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon." To appear in *Advances in Neural Information Processing Systems 5*. Morgan Kaufmann, 1993.

[48] D. Haussler, M. Kearns, M. Opper, and R. Schapire, "Estimating average-case learning curves using Bayesian, statistical physics and VC dimension methods." In J.E. Moody, S.J. Hanson, and R.P. Lippmann, editors, *Advances in Neural Information Processing Systems 4*, pages 855-862, 1992.

[49] S. Haykin, *Adaptive Filter Theory*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 1991.

[50] R. Hecht-Nielsen, *Neurocomputing*. Addison-Wesley, Menlo Park, CA, 1990.

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

[52] G.E. Hinton, "Connectionist learning procedures." Technical Report CMU-CS-87-115 (version 2), Carnegie-Mellon University, 1987.

[53] J.J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities." *Proceedings of the National Academy of Science USA*, 79:2554-2558, 1982.

[54] J.J. Hopfield, "Neurons with a graded response have collective computational properties like those of two-state neurons." *Proceedings of the National Academy of Science USA*, 81:3088-3092, May 1984.

[55] S.C. Huang and Y.F. Huang, "Bounds on the number of hidden neurons in multilayer perceptrons." *IEEE Transactions on Neural Networks*, 2(1):47-55, 1991.

[56] W.Y. Huang and R.P. Lippmann, "Neural net and traditional classifiers." In *Neural Information Processing Systems*, pp. 387-396, New York, NY, 1988. American Institute of Physics.

[57] D. Hush, C. Abdallah, and B. Horne, "Recursive neural networks for signal processing and control." In *Proceedings of the First IEEE-SP Workshop on Neural Networks for Signal Processing*, Princeton, NJ, pp. 523-532, 1991.

[58] D. Hush, J.M. Salas, and B. Horne, "Error surfaces for multi-layer perceptrons." *IEEE Transactions on Systems, Man and Cybernetics*, 22(5), 1992.

[59] D.R. Hush, "Classification with neural networks: A performance analysis." In *Proceedings of the IEEE International Conference on Systems Engineering*, pp. 277-280, 1989.

[60] D.R. Hush and J.M. Salas, "Improving the learning rate of back-propagation with the gradient reuse algorithm." In *Proceedings of the IEEE International Conference on Neural Networks*, volume 1, pages 441-448, 1988.

[61] W.T. Miller III, R.S. Sutton, and P.J. Werbos, editors. *Neural Networks for Control*. MIT Press, Cambridge, MA, 1990.

[62] A.C. Ivakhnenko, "Polynomial theory of complex systems." *IEEE Transactions on Systems, Man, and Cybernetics*, 1:364-378, 1971.

[63] R.A. Jacobs, "Increased rates of convergence through learning rate adaptation." *Neural Networks*, 1(4):295-308, 1988.

[64] A.K. Jain and R.C. Dubes, *Algorithms for Clustering Data*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[65] J.S. Judd, *Neural Network Design and the Complexity of Learning*. MIT Press, Cambridge, MA, 1990.

[66] T. Khanna, *Foundations of Neural Networks*. Addison-Wesley, Reading, MA, 1990.

[67] T. Kohonen, "An introduction to neural computing." *Neural Networks*, 1(1):3-16, 1988.

[68] B. Kosko, *Neural Networks and Fuzzy Systems: A Dynamical Systems Approach To Machine Intelligence*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[69] B. Kosko, *Neural Networks for Signal Processing*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[70] M.A. Kraaijveld and R.P.W. Duin, "Generalization capabilities of minimal kernel-based networks." In *Proceedings of the International Joint Conference on Neural Networks*, volume 1, pp. 843-848, 1991.

[71] A. Kuh and B.W. Dickinson, "Information capacity of associative memories." *IEEE Transactions on Information Theory*, 35:59-68, January 1989.

[72] S.Y. Kung, *Digital Neural Computing: from theory to application*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[73] K.J. Lang, A.H. Waibel, and G.E. Hinton, "A time-delay neural network architecture from isolated word recognition." *Neural Networks*, 3(1):23-44, 1990.

[74] A. Lapedes and R. Farber, "Nonlinear signal processing using neural networks: Prediction and signal modeling." Technical Report LA-UR-87-2662, Los Alamos National Laboratories, Los Alamos, New Mexico, 1987.

[75] Y. le Cun, "Generalization and network design strategies." In R. Pfeifer, Z. Schreter, F. Fogelman, and L. Steels, editors, *Connectionism in Perspective*. North Holland, Amsterdam, Netherlands, 1989.

[76] Y. le Cun, B. Boser, J.S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L.D. Jackel, "Backpropagation applied to handwritten zip code recognition." *Neural Computation*, 1(4):541-551, 1989.

[77] Y. le Cun, J.S. Denker, and S.A. Solla, "Optimal brain damage." In D. Touretzky, editor, *Advances in Neural Information Processing Systems 2* pp. 598-605. Morgan Kaufmann, 1990.

[78] S. Lee and R.M. Kil, "A gaussian potential function network with hierarchically self-organizing learning." *Neural Networks*, 4:207-224, 1991.

[79] Y. Lee and R.P. Lippmann, "Practical characteristics of neural network and conventional classifiers on artificial speech problems." In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pp. 168-177. Morgan Kaufmann, 1990.

[80] R. Lippmann, "A critical overview of neural network classifiers," In B.H. Juang, S.Y.Kung, and C.A. Kamm, editors, *Neural Networks for Signal Processing: Proceedings of the 1991 IEEE Workshop*, pp. 266-278. IEEE Press, 1991.

[81] R.P. Lippmann, "An introduction to computing with neural nets," *IEEE Acoustics, Speech and Signal Processing Magazine*, 4(2):4-22, April 1987.

[82] Y.D. Liu, G.Z. Sun, H.H. Chen, Y.C. Lee, and G.L. Giles, "Grammatical inference and neural network state machines." In M. Caudill, editor, *International Joint Conference on Neural Networks*, volume 1, pp. 285-288, Hillside, NJ, 1990. Erlbaum.

[83] D.J.C. MacKay, "A practical Bayesian framework for backprop networks." In J.E. Moody, S.J. Hanson, and R.P. Lippmann, editors, *Advances in Neural Information Processing Systems 4*, pp. 839-846, 1992.

[84] J. Makhoul, A. El-Jaroudi, and R. Schwartz, "Formation of disconnected decision regions with a single hidden layer." In *Proceedings of the International Joint Conference on Neural Networks*, volume 1, pages 455-460, 1989.

[85] C.L. Mallows, "Some comments on Cp." *Technometrics*, 15:661-675, 1973.

[86] R.J. Mammone and Y.Y. Zeevi, editors. *Neural Networks: Theory and Applications.* Academic Press, San Diego, CA, 1991.

[87] R.J. McEliece, E.C. Posner, E.R. Rodemich, and S.S. Venkatesh, "The capacity of the Hopfield associative memory," *IEEE Transactions on Information Theory*, 33:461-482, July 1987.

[88] M. Minsky, *Computation: Finite and infinite machines*. Prentice-Hall, Englewood Cliffs, 1967.

[89] M. Minsky and S. Papert, *Perceptrons*. MIT Press, Cambridge, MA, 1969.

[90] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, expanded edition, 1988.

[91] J. Moody and C.J. Darken, "Learning with localized receptive fields." In *Proceedings of the 1988 Connectionist Models Summer School*, pp. 133-143, 1988.

[92] J. Moody and C.J. Darken, "Fast learning in networks of locally-tuned processing units," *Neural Computation*, 1:281-293, 1989.

[93] J.E. Moody, "Note on generalization, regularization, and architecture selection in nonlinear learning systems." In B.H. Juang, S.Y. Kung, and C.A. Kamm, editors, *Neural Networks for Signal Processing: Proceedings of the 1991 IEEE Workshop*, pages 1-10. IEEE Press, 1991.

[94] J.E. Moody, "The effective number of parameters: An analysis of generalization and regularization in nonlinear learning systems." In J.E. Moody, S.J. Hanson, and R.P. Lippmann, editors, *Advances in Neural Information Processing Systems* 4, pages 847-854, 1992.

[95] D.P. Morgan and C.L. Scofield, *Neural Networks and Speech Processing*. Kluwer Academic Publishers, 1991.

[96] S. Muroga, *Threshold Logic and Its Applications*. Wiley, New York, NY, 1971.

[97] M.T. Musavi, W. Ahmed, K.H. Chan, K.B. Faris, and D.M. Hummels, "On the training of radial basis function classifiers," *Neural Networks*, 5:595-603, 1992.

[98] K.S. Narendra and K. Parthasarathy, "Identification and control of dynamical systems using neural networks," *IEEE Transactions on Neural Networks*, 1:4-27, March 1990.

[99] K.S. Narendra and K. Parthasarathy, "Gradient methods for the optimization of dynamical systems containing neural networks," *IEEE Transactions on Neural Networks*, 2:252-262, March 1991.

[100] B.K. Natarajan, *Machine learning: A theoretical approach*. Morgan Kaufmann, San Mateo, CA, 1991.

[101] K. Ng and R.P. Lippmann, "A comparative study of the practical characteristics of neural network and conventional classifiers." In R.P. Lippmann, J.E. Moody, and D.S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pp. 970- 976. Morgan Kaufmann, 1991.

[102] S.J. Nowlan and G.E. Hinton, "Adaptive soft weight tying using Gaussian mixtures." In J.E. Moody, S.J. Hanson, and R.P. Lippmann, editors, *Advances in Neural Information Processing Systems* 4, pp. 993-1000. Morgan Kaufmann, 1992.

[103] S.J. Nowlan and G.E. Hinton, "Simplifying neural networks by soft weight sharing." *Neural Computation*, 4(4):473-493, 1992.

[104] A. Oppenheim and R.W. Schafer, *Digital Signal Processing*. Prentice-Hall, Englewood Cliffs, NJ, 1975.

[105] Y.H. Pao, *Adaptive Pattern Recognition and Neural Networks*. Addison-Wesley, Reading, MA, 1989.

[106] D.B. Parker, "Learning-logic." Technical Report TR-47, Center for Comp. Res. in Econ. and Man., MIT, Cambridge, MA, April 1985.

[107] B.A. Pearlmutter, "Learning state space trajectories in recurrent neural networks," *Neural Computation*, 1(2):263-269, 1989.

[108] J.A. Anderson A. Pellionisz and E. Rosenfeld, *Neurocomputing 2*. MIT Press, Cambridge, MA, 1990.

[109] F.J. Pineda, "Dynamics and architecture for neural computation," *Journal of Complexity*, 4:216-245, 1988.

[110] D.C. Plaut, S.J. Nowlan, and G.E. Hinton, "Experiments on learning back propagation." Technical Report CMU-CS-86-126, Carnegie-Mellon University, Pittsburgh, PA, 1986.

[111] T. Poggio and F. Girosi, "A theory of networks for approximating and learning." Artificial Intelligence Lab. Memo 1140, MIT, 1989.

[112] J . B. Pollack, "Implications of recursive distributed representations." In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, 1, pp. 527-536. Morgan Kaufmann, 1989.

[113] M.J.D. Powell, "Radial basis functions for multivariate interpolation: A review." Technical Report DAMPT 1985/NA12, Dept. of App. Math. and Theor. Physics, Cambridge University, Cambridge, England, 1985.

[114] M.D. Richard and R.P Lippmann, "Neural network classifiers estimate Bayesian a posteriori probabilities." *Neural Computation*, 3(4):461-483, 1991.

[115] J. Rissanen, "Modeling by shortest data description," *Automatica*, 14:465-471, 1978.

[116] J. Rissanen, "Stochastic complexity and modeling," *Annals of Statistics*, 14:1080-1100, 1986.

[117] A.J. Robinson and F. Fallside, "Static and dynamic error propagation networks with application to speech coding." In D.Z. Anderson, editor, *Neural Information Processing Systems*, pp. 632-641, New York, NY, 1988. American Institute of Physics.

[118] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain." *Psychological Review*, 65:386-408, 1958.

[119] D.E. Rumelhart, G.E. Hinton, and R.J. Williams, "Learning internal representations by error propagation." In D.E. Rumelhart and J.L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, pp. 318-362. MIT Press, Cambridge, MA, 1986.

[120] D.E. Rumelhart and J.L. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*, volume 1. MIT Press, Cambridge, MA, 1986.

[121] D.E. Rumelhart and J.L. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Psychological and Biological Models*, volume 2. MIT Press, Cambridge, MA, 1986.

[122] Y.M. Sandler, "Model of neural networks with selective memorization and chaotic behavior." *Physics Letters*, A, 144:462-466, 1990.

[123] E.L. Schwartz, editor. *Computational Neuroscience*. MIT Press, Cambridge, MA, 1990.

[124] T.J. Sejnowski and C.R. Rosenberg, "NETtalk: A parallel network that learns to read aloud." Technical Report JHU/EECS-86/01, Johns Hopkins University, Baltimore, MD, 1986.

[125] H. Siegelman and E.D. Sontag, "Neural networks are universal computing devices." Technical Report SYCON-91-08, Rutgers Center for Systems and Control, 1991.

[126] P.K. Simpson, *Artificial Neural Systems*. Pergamon Press, Elmsford, NY, 1989.

[127] K.Y. Siu, V. Roychowdhury, and T. Kailath, "Depth-size tradeoffs for neural computation." Technical report, Information Systems Laboratory, Stanford University, 1990.

[128] E.D. Sontag, "Sigmoids distinguish more efficiently than heavisides." *Neural Computation*, 1:470-472, 1989.

[129] D.F. Specht, "Probabilistic neural networks." *Neural Networks*, 3:109-118, 1990.

[130] S.D. Stearns and D.R. Hush, *Digital Signal Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[131] G.Z. Sun, H.H. Chen, Y.C. Lee, and C.L. Giles, "Turing equivalence of neural networks with second order connection weights." In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, pp. 357-362, 1991.

[132] R.S. Sutton, "Learning to predict by methods of temporal differences." *Machine Learning*, 3:9-44, 1988.

[133] G. Tesauro, "Practical issues in temporal difference learning." In J. Moody, S.J. Hanson, and R.P. Lippmann, editors, *Advances in Neural Information Processing Systems 4*, pp. 259-266. Morgan Kaufmann, 1992.

[134] J.T. Tou and R.C. Gonzalez, *Pattern Recognition Principles*. Addison-Wesley, Reading, MA, 1974.

[135] V.N. Vapnik and A.Ya. Chervonenkis. "On the uniform convergence of relative frequencies of events to their probabilities." *Theory of Probability and its Applications*, 16(2):264-280, 1971.

[136] S.S. Venkatesh and D. Psaltis, "Linear and logarithmic capacities in associative neural networks." *IEEE Transactions on Information Theory*, 35:558-568, May 1989.

[137] M. Vidyasagar. *Nonlinear Systems Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[138] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. Lang, "Phoneme recognition using time-delay neural networks," *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37(3) 328-339, 1989.

[139] P.D. Wasserman, *Neural Computing: Theory and Practice*. Van Nostrand Reinhold, New York, NY, 1989.

[140] R.L. Watrous, "Learning algorithms for connectionist networks: applied gradient methods of nonlinear optimization." In *Proceedings IEEE 1st International Conference on Neural Networks*, volume 2, pages 619-628, 1987.

[141] R.L. Watrous and G.M. Kuhn, "Induction of finite-state languages using second-order recurrent networks." *Neural Computation*, 4(3):406-414, 1992.

[142] A.S. Weigend, B.A. Huberman, and D.E. Rumelhart, "Predicting sunspots and exchange rates with connectionist networks." In M. Casdagli and S. Eubank, editors, *Nonlinear Modeling and Forcasting, SFI Studies in the Sciences of Complexity*, volume 12, Addison-Wesley, 1991.

[143] A.S. Weigend, D.E. Rumelhart, and B.A. Huberman, "Back-propagation, weight elimination and time series prediction." In *Proceedings of the 1990 Connectionist Models Summer School*, pp. 65-80. Morgan Kaufmann, 1990.

[144] P.J. Werbos, "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences." Doctoral Dissertation, Applied Mathematics, Harvard University, Boston, MA, November 1974.

[145] D. Wettschereck and T. Dietterich, "Improving the performance of radial basis function networks by learning center locations." In J.E. Moody, S.J. Hanson, and R.P. Lippmann, editors, *Advances in Neural Information Processing Systems* 4, pp. 1133-1140, 1992.

[146] H. White, "Learning in artificial neural networks: A statistical perspective." *Neural Computation*, 1(4):425-464, 1989.

[147] P. Whittle, *Prediction and regulation by linear least-square methods*. Van Nostrand, Princeton, N.J., 1963.

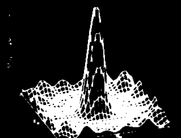
[148] B. Widrow. "ADALINE and MADALINE - 1963." In *Proceedings IEEE 1st International Conference on Neural Networks*, volume 1, pp. 143-157, 1987. Plenary Speech.

[149] B. Widrow and M.E. Hoff, "Adaptive switching circuits." In *1960 IRE WESCON Convention Record*, pages 96-104, New York, NY, 1960.

[150] R.J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks." *Neural Computation*, 1(2):270-280, 1989.

[151] R.J. Williams and D. Zipser, "Gradient-based learning algorithms for recurrent connectionist networks." Technical Report NU-CCS-90-9, College of Computer Science, Northeastern University, 1990.

[152] J.M. Zurada, *Introduction to Artificial Neural Systems*, West Publishing Company, St. Paul, MN, 1992.



sigx™

A General Purpose Signal Processing Package

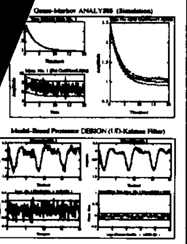
- Customizable DSP/Graphics Environment
- Mouse-Driven/Pull-Down Menu/Cmd. Proc.
- Manipulative Signal/Image Database
- Over 200 1D/2D DSP Commands
- SIGtutor/SIGtools Developmt. Optns.

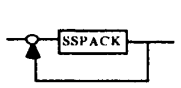
Signal Processing Operations Include: Windowing convolution, Fourier transforms, interpolation, decimation, digital filtering, linear systems, simulation, correlation, ensemble operations, spectral estimation, coherence, parametric and adaptive processing, deconvolution, identification and more.

Features Include: Surface (3D) plots, contour plots, multichannel signal and complex operations, 2D Fourier transforms, additional algorithms and more

- Interactive environment /pull-down menu and cmd. proc.
- Powerful statistics/graphics processor for analysis / display
- Unique model-based signal processing using powerful UD-Kalman filters

- Built-in Sample Problems
- Graphics Include: Multiple Viewports, Families of Curves





for use with your *MATLAB™

SSPACK™
State-Space Systems Software Package

SSPACK_PC™
SSPACK on Personal Computers

Available On: SUN, DECstation, VAX, HP9000, SGI, IBM/RS/6000

*MATLAB is a trademark of The Math Works, Inc

Techni Soft
P.O. Box 2525, Livermore, CA 94551, (510) 443-7213, Fax (510) 743-1145

Circle Reader Service Number 8

Connection Science

CALL FOR PAPERS FOR SPECIAL ISSUE:

ARCHITECTURES FOR INTEGRATING NEURAL AND SYMBOLIC PROCESSES

This special issue is intended to address the cognitive architectural aspects of the integration of neural and symbolic processes. Papers can be either theoretical or experimental in scope, and can comment on the current state of affairs and address what advances are necessary so that continued progress can be made.

For submission details and further guidance contact:

Dr. Lawrence Bookman	*or*	Prof. Ron Sun
Sun Microsystems Laboratories		Department of Computer Science
Two Federal Street		The University of Alabama
Billerica, MA 01821, USA		Tuscaloosa, AL 35487, USA
Net: lbookman@east.sun.com		Net: rsun@athos.cs.ua.edu

FOR A SAMPLE COPY OF THE JOURNAL CONTACT:

carfax publishing company

Carfax Publishing Company
P.O. Box 25, Abingdon, Oxfordshire
OX14 3UE, UK. Fax 0235 553559 (UK)
+44 235 553559 (overseas)

Reader Service Number 21