1

Introduction to Neural Networks for Signal Processing

Yu Hen Hu
University of Wisconsin

Jenq-Neng Hwang
University of Washington

1.1 Introduction

The theory and design of artificial neural networks have advanced significantly during the past 20 years. Much of that progress has a direct bearing on signal processing. In particular, the nonlinear nature of neural networks, the ability of neural networks to learn from their environments in supervised as well as unsupervised ways, as well as the universal approximation property of neural networks make them highly suited for solving difficult signal processing problems.

From a signal processing perspective, it is imperative to develop a proper understanding of basic neural network structures and how they impact signal processing algorithms and applications. A challenge in surveying the field of neural network paradigms is to identify those neural network structures that have been successfully applied to solve real world problems from those that are still under development or have difficulty scaling up to solve realistic problems. When dealing with signal processing applications, it is critical to understand the nature of the problem formulation so that the most appropriate neural network paradigm can be applied. In addition, it is also important to assess the impact of neural networks on the performance, robustness, and cost-effectiveness of signal processing systems and develop methodologies for integrating neural networks with other signal processing algorithms. Another important issue is how to evaluate neural network paradigms, learning algorithms, and neural network structures and identify those that do and do not work reliably for solving signal processing problems.

This chapter provides an overview of the topic of this handbook — neural networks for signal processing. The chapter first discusses the definition of a neural network for signal processing and why it is important. It then surveys several modern neural network models that have found successful signal processing applications. Examples are cited relating to how to apply these nonlinear

0-8493-2359-2/01/$0.00+$1.50

computation paradigms to solve signal processing problems. Finally, this chapter highlights the remaining contents of this book.

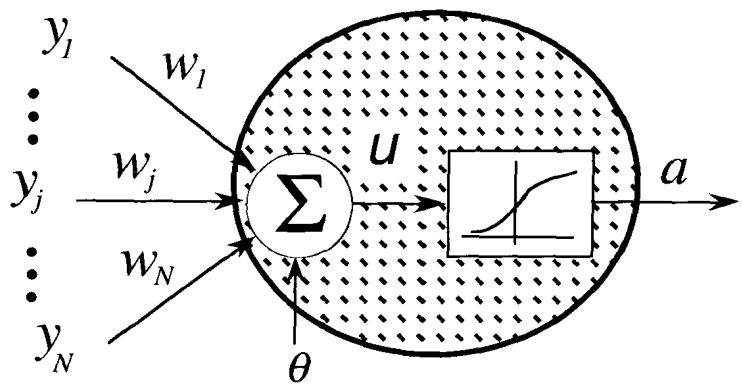
1.2 Artificial Neural Network (ANN) Models — An Overview

1.2.1 Basic Neural Network Components

A neural network is a general mathematical computing paradigm that models the operations of biological neural systems. In 1943, McCulloch, a neurobiologist, and Pitts, a statistician, published a seminal paper titled "A logical calculus of ideas imminent in nervous activity" in *Bulletin of Mathematical Biophysics* [1]. This paper inspired the development of the modern digital computer, or the electronic brain, as John von Neumann called it. At approximately the same time, Frank Rosenblatt was also motivated by this paper to investigate the computation of the eye, which eventually led to the first generation of neural networks, known as the perceptron [2]. This section provides a brief overview of ANN models. Many of these topics will be treated in greater detail in later chapters. The purpose of this chapter, therefore, is to highlight the basic concept of these neural network models to prepare the readers for later chapters.

1.2.1.1 McCulloch and Pitts' Neuron Model

Among numerous neural network models that have been proposed over the years, all share a common building block known as a neuron and a networked interconnection structure. The most widely used neuron model is based on McCulloch and Pitts' work and is illustrated in Figure 1.1.



1.1 McCulloch and Pitts' neuron model.

In Figure 1.1, each neuron consists of two parts: the net function and the activation function. The net function determines how the network inputs $\{y_j;\ 1 \leq j \leq N\}$ are combined inside the neuron. In this figure, a weighted linear combination is adopted:

$$u = \sum_{j=1}^{N} w_j y_j + \theta \tag{1.1}$$

$\{w_j;\ 1 \leq j \leq N\}$ are parameters known as synaptic weights. The quantity θ is called the bias (or threshold) and is used to model the threshold. In the literature, other types of network input combination methods have been proposed as well. They are summarized in Table 1.1.

TABLE 1.1 Summary of Net Functions

Net Functions	Formula	Comments
Linear	$u = \sum_{j=1}^{N} w_j y_j + \theta$	Most commonly used
Higher order (2nd order formula exhibited)	$u = \sum_{j=1}^{N} \sum_{k=1}^{N} w_{jk} y_j y_k + \theta$	u_i is a weighted linear combination of higher order polynomial terms of input variable. The number of input terms equals N^d, where d is the order of the polynomial
Delta ($\sum - \prod$)	$u = \prod_{j=1}^{N} w_j y_j$	Seldom used

The output of the neuron, denoted by a_i in this figure, is related to the network input u_i via a linear or nonlinear transformation called the activation function:

$$a = f(u) . \tag{1.2}$$

In various neural network models, different activation functions have been proposed. The most commonly used activation functions are summarized in Table 1.2.

TABLE 1.2 Neuron Activation Functions

Activation Function	Formula $a = f(u)$	Derivatives $\frac{df(u)}{du}$	Comments
Sigmoid	$f(u) = \frac{1}{1+e^{-u/T}}$	$f(u)[1 - f(u)]/T$	Commonly used; derivative can be computed from $f(u)$ directly.
Hyperbolic tangent	$f(u) \tanh\left(\frac{u}{T}\right)$	$\left(1 - [f(u)]^2\right)/T$	T = temperature parameter
Inverse tangent	$f(u) = \frac{2}{\pi} \tan^{-1}\left(\frac{u}{T}\right)$	$\frac{2}{\pi T} \cdot \frac{1}{1+(u/T)^2}$	Less frequently used
Threshold	$f(u) = \begin{cases} 1 & u > 0; \\ -1 & u < 0. \end{cases}$	Derivatives do not exist at $u = 0$	
Gaussian radial basis	$f(u) = \exp\left[-\|u - m\|^2/\sigma^2\right]$	$-2(u - m) \cdot f(u)/\sigma^2$	Used for radial basis neural network; m and σ^2 are parameters to be specified
Linear	$f(u) = au + b$	a	

Table 1.2 lists both the activation functions as well as their derivatives (provided they exist). In both sigmoid and hyperbolic tangent activation functions, derivatives can be computed directly from the knowledge of $f(u)$.

1.2.1.2 Neural Network Topology

In a neural network, multiple neurons are interconnected to form a network to facilitate distributed computing. The configuration of the interconnections can be described efficiently with a directed graph. A directed graph consists of nodes (in the case of a neural network, neurons, as well as external inputs) and directed arcs (in the case of a neural network, synaptic links).

The topology of the graph can be categorized as either acyclic or cyclic. Refer to Figure 1.2a; a neural network with acyclic topology consists of no feedback loops. Such an acyclic neural network is often used to approximate a nonlinear mapping between its inputs and outputs. As shown in Figure 1.2b, a neural network with cyclic topology contains at least one cycle formed by directed arcs. Such a neural network is also known as a recurrent network. Due to the feedback loop, a recurrent network leads to a nonlinear dynamic system model that contains internal memory. Recurrent neural networks often exhibit complex behaviors and remain an active research topic in the field of artificial neural networks.

(a) Acyclic topology (b) Cyclic topology

1.2 Illustration of (a) an acyclic graph and (b) a cyclic graph. The cycle in (b) is emphasized with thick lines.

1.2.2 Multilayer Perceptron (MLP) Model

The multilayer perceptron [3] is by far the most well known and most popular neural network among all the existing neural network paradigms. To introduce the MLP, let us first discuss the perceptron model.

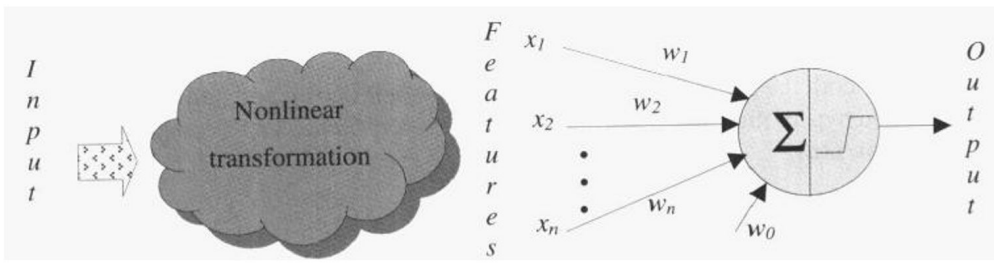
1.2.2.1 Perceptron Model

An MLP is a variant of the original perceptron model proposed by Rosenblatt in the 1950s [2]. In the perceptron model, a single neuron with a linear weighted net function and a threshold activation function is employed. The input to this neuron $\underline{x} = (x_1, x_2, \ldots, x_n)$ is a feature vector in an n-dimensional feature space. The net function $u(\underline{x})$ is the weighted sum of the inputs:

$$u(\underline{x}) = w_0 + \sum_{i=1}^{n} w_i x_i \tag{1.3}$$

and the output $y(\underline{x})$ is obtained from $u(\underline{x})$ via a threshold activation function:

$$y(\underline{x}) = \begin{cases} 1 & u(\underline{x}) \geq 0 \\ 0 & u(\underline{x}) < 0 \,. \end{cases} \tag{1.4}$$



1.3 A perceptron neural network model.

The perceptron neuron model can be used for detection and classification. For example, the weight vector $\underline{w} = (w_1, w_2, \ldots, w_n)$ may represent the template of a certain target. If the input feature vector $\underline{x}$ closely matches $\underline{w}$ such that their inner product exceeds a threshold $-w_0$, then the output will become $+1$, indicating the detection of a target.

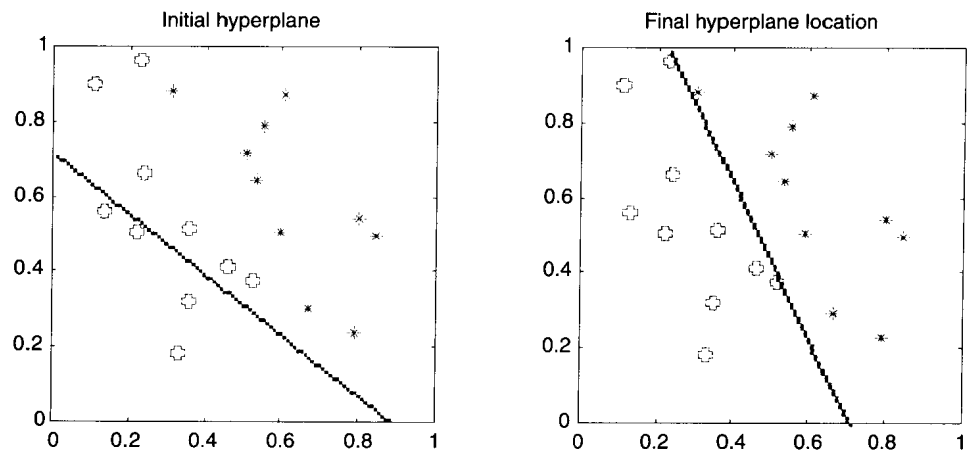
The weight vector $\underline{w}$ needs to be determined in order to apply the perceptron model. Often, a set of training samples $\{(\underline{x}(i), d(i)); i \in I_r\}$ and testing samples $\{(\underline{x}(i), d(i)); i \in I_t\}$ are given. Here, $d(i) (\in \{0, 1\})$ is the desired output value of $y(\underline{x}(i))$ if the weight vector $\underline{w}$ is chosen correctly, and I_r and I_t are disjoined index sets. A sequential online perceptron learning algorithm can be applied to iteratively estimate the correct value of $\underline{w}$ by presenting the training samples to the perceptron

neuron in a random, sequential order. The learning algorithm has the following formulation:

$$\underline{w}(k+1) = \underline{w}(k) + \eta(d(k) - y(k))\underline{x}(k) \tag{1.5}$$

where $y(k)$ is computed using Equations (1.3) and (1.4). In Equation (1.5), the learning rate $\eta(0 < \eta < 1/|\underline{x}(k)|_{\max})$ is a parameter chosen by the user, where $|\underline{x}(k)|_{\max}$ is the maximum magnitude of the training samples $\{\underline{x}(k)\}$. The index k is used to indicate that the training samples are applied sequentially to the perceptron in a random order. Each time a training sample is applied, the corresponding output of the perceptron $y(k)$ is to be compared with the desired output $d(k)$. If they are the same, meaning the weight vector $\underline{w}$ is correct for this training sample, the weights will remain unchanged. On the other hand, if $y(k) \neq d(k)$, then $\underline{w}$ will be updated with a small step along the direction of the input vector $\underline{x}(k)$. It has been proven that if the training samples are linearly separable, the perceptron learning algorithm will converge to a feasible solution of the weight vector within a finite number of iterations. On the other hand, if the training samples are not linearly separable, the algorithm will not converge with a fixed, nonzero value of η.

MATLAB Demonstration Using MATLAB m-files `perceptron.m`, `datasepf.m`, and `sline.m`, we conducted a simulation of a perceptron neuron model to distinguish two separable data samples in a two-dimensional unit square. Sample results are shown in Figure 1.4.



1.4 Perceptron simulation results. The figure on the left-hand side depicts the data samples and the initial position of the separating hyperplane, whose normal vector contains the weights to the perceptron. The right-hand side illustrates that the learning is successful as the final hyperplane separates the two classes of data samples.

1.2.2.1.1 Applications of the Perceptron Neuron Model

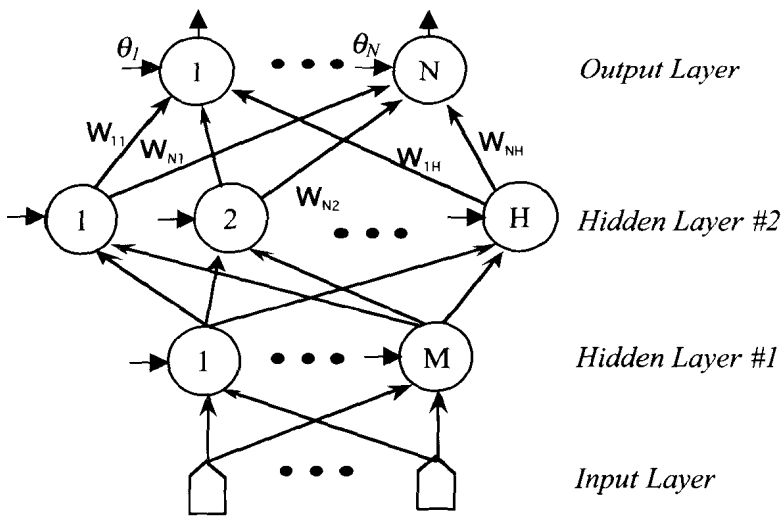
There are several major difficulties in applying the perceptron neuron model to solve real world pattern classification and signal detection problems:

1. The nonlinear transformation that extracts the appropriate feature vector x is not specified.
2. The perceptron learning algorithm will not converge for a fixed value of learning rate η if the training feature patterns are not linearly separable.
3. Even though the feature patterns are linearly separable, it is not known how long it takes for the algorithm to converge to a weight vector that corresponds to a hyperplane that separates the feature patterns.

1.2.2.2 Multilayer Perceptron

A multilayer perceptron (MLP) neural network model consists of a feed-forward, layered network of McCulloch and Pitts' neurons. Each neuron in an MLP has a nonlinear activation function that is often continuously differentiable. Some of the most frequently used activation functions for MLP include the sigmoid function and the hyperbolic tangent function.

A typical MLP configuration is depicted in Figure 1.5. Each circle represents an individual neuron. These neurons are organized in layers, labeled as the hidden layer #1, hidden layer #2, and the output layer in this figure. While the inputs at the bottom are also labeled as the input layer, there is usually no neuron model implemented in that layer. The name *hidden layer* refers to the fact that the output of these neurons will be fed into upper layer neurons and, therefore, is hidden from the user who only observes the output of neurons at the output layer. Figure 1.5 illustrates a popular configuration of MLP where interconnections are provided only between neurons of successive layers in the network. In practice, any acyclic interconnections between neurons are allowed.



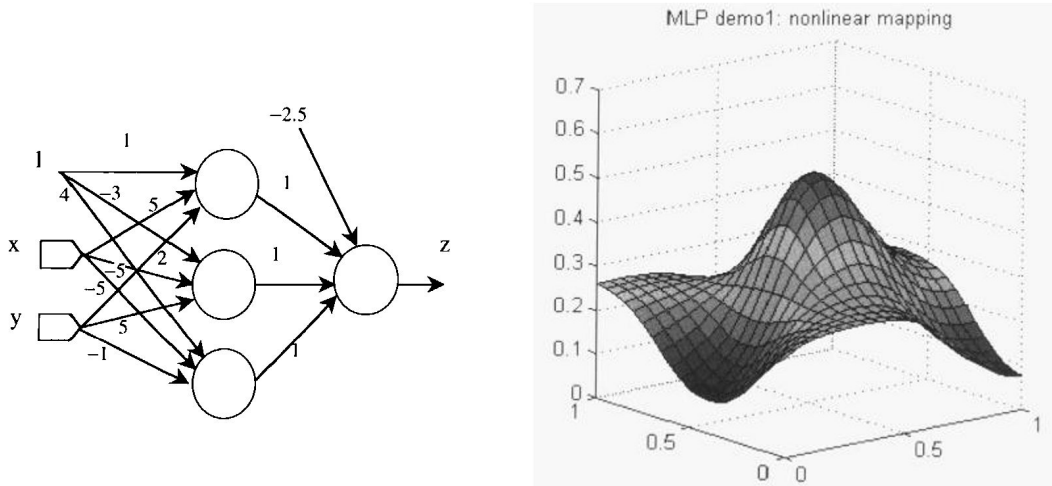
1.5 A three-layer multilayer perceptron configuration.

An MLP provides a nonlinear mapping between its input and output. For example, consider the following MLP structure (Figure 1.6) where the input samples are two-dimensional grid points, and the output is the z-axis value. Three hidden nodes are used, and the sigmoid function has a parameter $T = 0.5$. The mapping is plotted on the right side of Figure 1.6. The nonlinear nature of this mapping is quite clear from the figure. The MATLAB m-files used in this demonstration are `mlpdemo1.m` and `mlp2.m`.

It has been proven that with a sufficient number of hidden neurons, an MLP with as few as two hidden layer neurons is capable of approximating an arbitrarily complex mapping within a finite support [4].

1.2.2.3 Error Back-Propagation Training of MLP

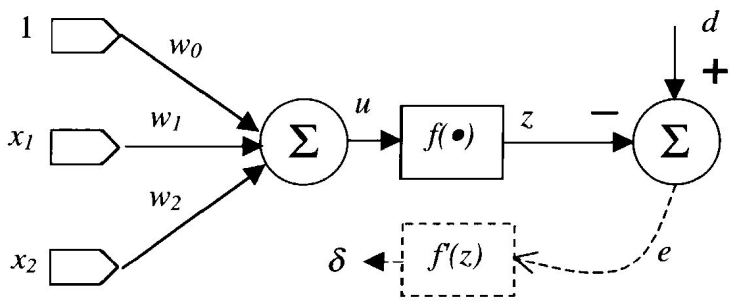
A key step in applying an MLP model is to choose the weight matrices. Assuming a layered MLP structure, the weights feeding into each layer of neurons form a weight matrix of that layer (the input layer does not have a weight matrix as it contains no neurons). The values of these weights are found using the error back-propagation training method.



1.6 Demonstration of nonlinear mapping property of MLP.

1.2.2.3.1 *Finding the Weights of a Single Neuron MLP*

For convenience, let us first consider a simple example consisting of a single neuron to illustrate this procedure. For clarity of explanation, Figure 1.7 represents the neuron in two separate parts: a summation unit to compute the net functions u, and a nonlinear activation function $z = f(u)$. The



1.7 MLP example for back-propagation training — single neuron case.

output z is to be compared with a desired target value d, and their difference, the error $e = d - z$, will be computed. There are two inputs $[x_1 \quad x_2]$ with corresponding weights w_1 and w_2. The input labeled with a constant 1 represents the bias term θ shown in Figures 1.1 and 1.5 above. Here, the bias term is labeled w_0. The net function is computed as:

$$u = \sum_{i=0}^{2} w_i x_i = \mathbf{W}\mathbf{x} \tag{1.6}$$

where $x_0 = 1$, $\mathbf{W} = [w_0 \, w_1 \, w_2]$ is the weight matrix, and $\mathbf{x} = [1 \, x_1 \, x_2]^T$ is the input vector.

Given a set of training samples $\{(\mathbf{x}(k), d(k)); 1 \leq k \leq K\}$, the error back-propagation training begins by feeding all K inputs through the MLP network and computing the corresponding output $\{z(k); 1 \leq k \leq K\}$. Here we use an initial guess for the weight matrix $\mathbf{W}$. Then a sum of square

error will be computed as:

$$E = \sum_{k=1}^{K}[e(k)]^2 = \sum_{k=1}^{K}[d(k) - z(k)]^2 = \sum_{k=1}^{K}[d(k) - f(\mathbf{W}\mathbf{x}(k))]^2 \ . \tag{1.7}$$

The objective is to adjust the weight matrix $\mathbf{W}$ to minimize the error E. This leads to a nonlinear least square optimization problem. There are numerous nonlinear optimization algorithms available to solve this problem. Basically, these algorithms adopt a similar iterative formulation:

$$\mathbf{W}(t+1) = \mathbf{W}(t) + \Delta\mathbf{W}(t) \tag{1.8}$$

where $\Delta\mathbf{W}(t)$ is the correction made to the current weights $\mathbf{W}(t)$. Different algorithms differ in the form of $\Delta\mathbf{W}(t)$. Some of the important algorithms are listed in Table 1.3.

TABLE 1.3 Iterative Nonlinear Optimization Algorithms to Solve for MLP Weights

Algorithm	$\Delta\mathbf{W}(t)$	Comments
Steepest descend gradient method	$= -\eta \mathbf{g}(t) = -\eta \ dE/d\mathbf{W}$	$\mathbf{g}$ is known as the gradient vector. η is the step size or learning rate. This is also known as error back-propagation learning.
Newton's method	$= -\boldsymbol{H}^{-1}\mathbf{g}(t)$	$\mathbf{H}$ is known as the Hessian matrix. There are several different ways to estimate it.
	$= -\left[d^2E/d\mathbf{W}^2\right]^{-1}(dE/d\mathbf{W})$	
Conjugate-Gradient method	$= \eta\mathbf{p}(t)$ where $\mathbf{p}(t+1) = -\mathbf{g}(t+1) + \beta\ \mathbf{p}(t)$	

This section focuses on the steepest descend gradient method that is also the basis of the error back-propagation learning algorithm. The derivative of the scalar quantity E with respect to individual weights can be computed as follows:

$$\frac{\partial E}{\partial w_i} = \sum_{k=1}^{K}\frac{\partial[e(k)]^2}{\partial w_i} = \sum_{k=1}^{K} 2[d(k) - z(k)]\left(-\frac{\partial z(k)}{\partial w_i}\right) \qquad \text{for } i = 0, 1, 2 \tag{1.9}$$

where

$$\frac{\partial z(k)}{\partial w_i} = \frac{\partial f(u)}{\partial u}\frac{\partial u}{\partial w_i} = f'(u)\frac{\partial}{\partial w_i}\left(\sum_{j=0}^{2} w_j x_j\right) = f'(u)x_i \tag{1.10}$$

Hence,

$$\frac{\partial E}{\partial w_i} = -2\sum_{k=1}^{K}[d(k) - z(k)]f'(u(k))x_i(k) \ . \tag{1.11}$$

With $\delta(k) = [d(k) - z(k)]f'(u(k))$, the above equation can be expressed as:

$$\frac{\partial E}{\partial w_i} = -2\sum_{k=1}^{K}\delta(k)x_i(k) \tag{1.12}$$

$\delta(k)$ is the error signal $e(k) = d(k) - z(k)$ modulated by the derivative of the activation function $f'(u(k))$ and hence represents the amount of correction needed to be applied to the weight w_i for the

given input $x_i(k)$. The overall change Δw_i is thus the sum of such contribution over all K training samples. Therefore, the weight update formula has the format of:

$$w_i(t+1) = w_i(t) + \eta \sum_{k=1}^{K} \delta(k) x_i(k) \; . \tag{1.13}$$

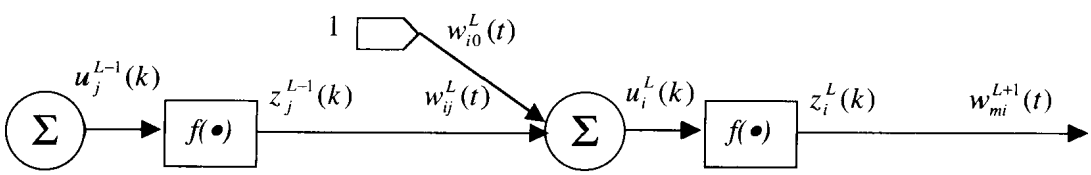
If a sigmoid activation function as defined in Table 1.1 is used, then $\delta(k)$ can be computed as:

$$\delta(k) = \frac{\partial E}{\partial u} = [d(k) - z(k)] \cdot z(k) \cdot [1 - z(k)] \; . \tag{1.14}$$

Note that the derivative $f'(u)$ can be evaluated exactly without any approximation. Each time the weights are updated is called an epoch. In this example, K training samples are applied to update the weights once. Thus, we say the epoch size is K. In practice, the epoch size may vary between one and the total number of samples.

1.2.2.3.2 Error Back-Propagation in a Multiple Layer Perceptron

So far, this chapter has discussed how to adjust the weights (training) of an MLP with a single layer of neurons. This section discusses how to perform training for a multiple layer MLP. First, some new notations are adopted to distinguish neurons at different layers. In Figure 1.8, the net-function and output corresponding to the kth training sample of the jth neuron of the $(L-1)$th are denoted by $u_j^{L-1}(k)$ and $z_j^{L-1}(k)$, respectively. The input layer is the *zero*th layer. In particular, $z_j^0(k) = x_j(k)$. The output is fed into the ith neuron of the Lth layer via a synaptic weight denoted by $w_{ij}^L(t)$ or, for simplicity, w_{ij}^L, since we are concerned with the weight update formulation within a single training epoch.



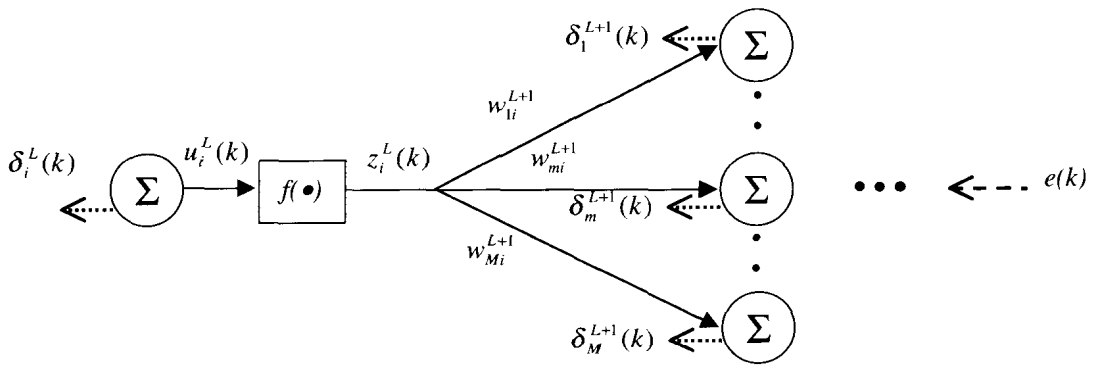
1.8 Notations used in a multiple-layer MLP neural network model.

To derive the weight adaptation equation, $\partial E / \partial w_{ij}^L$ must be computed:

$$\begin{aligned} \frac{\partial E}{\partial w_{ij}^L} &= -2 \sum_{k=1}^{K} \frac{\partial E}{\partial u_i^L(k)} \cdot \frac{\partial u_i^L(k)}{\partial w_{ij}^L} = -2 \sum_{k=1}^{K} \left[\delta_i^L(k) \cdot \frac{\partial}{\partial w_{ij}^L} \sum_m w_{im}^L z_m^{L-1}(k) \right] \\ &= -2 \sum_{k=1}^{K} \delta_i^L(k) \cdot z_j^{L-1}(k) \; . \end{aligned} \tag{1.15}$$

In Equation (1.15), the output $z_j^{L-1}(k)$ can be evaluated by applying the kth training sample $\mathbf{x(k)}$ to the MLP with weights fixed to w_{ij}^L. However, the delta error term $\delta_i^L(k)$ is not readily available and has to be computed.

Recall that the delta error is defined as $\delta_i^L(k) = \partial E / \partial u_i^L(k)$. Figure 1.9 is now used to illustrate how to iteratively compute $\delta_i^L(k)$ from $\delta_m^{L+1}(k)$ and weights of the $(L+1)$th layer.



1.9 Illustration of how the error back-propagation is computed.

Note that $z_i^L(k)$ is fed into all M neurons in the $(L+1)$th layer. Hence:

$$\delta_i^L(k) = \frac{\partial E}{\partial u_i^L(k)} = \sum_{m=1}^{M} \frac{\partial E}{\partial u_m^{L+1}(k)} \cdot \frac{\partial u_m^{L+1}(k)}{\partial u_i^L(k)} = \sum_{m=1}^{M} \left[\delta_m^{L+1}(k) \cdot \frac{\partial}{\partial u_i^L(k)} \sum_{j=1}^{J} w_{mj}^L f\left(u_j^L(k)\right) \right]$$

$$= f'\left(u_i^L(k)\right) \cdot \sum_{m=1}^{M} \delta_m^{L+1}(k) \cdot w_{mi}^L \,. \tag{1.16}$$

Equation (1.16) is the error back-propagation formula that computes the delta error from the output layer back toward the input layer, in a layer-by-layer manner.

1.2.2.3.3 Weight Update Formulation with Momentum and Noise

Given the delta error, the weights will be updated according to a modified formulation of Equation (1.13):

$$w_{ij}^L(t+1) = w_{ij}^L(t) + \eta \cdot \sum_{k=1}^{K} \delta_i^L(k) z_j^{L-1}(k) + \mu \left[w_{ij}^L(t) - w_{ij}^L(t-1) \right] + \varepsilon_{ij}^L(t) \,. \tag{1.17}$$

On the right hand side of Equation (1.17), the second term is the gradient of the mean square error with respect to w_{ij}^L. The third term is known as a momentum term. It provides a mechanism to adaptively adjust the step size. When the gradient vectors in successive epochs point to the same direction, the effective step size will increase (gaining momentum). When successive gradient vectors form a zigzag search pattern, the effective gradient direction will be regulated by this momentum term so that it helps minimize the mean-square error.

There are two parameters that must be chosen: the learning rate, or step size η, and the momentum constant μ. Both of these parameters should be chosen from the interval of [0 1]. In practice, η often assumes a smaller value, e.g., $0 < \eta < 0.3$, and μ usually assumes a larger value, e.g., $0.6 < \mu < 0.9$.

The last term in Equation (1.17) is a small random noise term that will have little effect when the second or the third terms have larger magnitudes. When the search reaches a local minimum or a plateau, the magnitude of the corresponding gradient vector or the momentum term is likely to diminish. In such a situation, the noise term can help the learning algorithm leap out of the local minimum and continue to search for the globally optimal solution.

1.2.2.3.4 Implementation of the Back-Propagation Learning Algorithm

With the new notations and the error back-propagation formula, the back-propagation training algorithm for MLP can be summarized below in the MATLAB m-file format:

Algorithm Listing: Back-Propagation Training Algorithm for MLP

```
% configure the MLP network and learning parameters.
bpconfig;      % call mfile bpconfig.m

% BP iterations begins
while not_converged==1,
 % start a new epoch
 % Randomly select K training samples from the training set.
 [train,ptr,train0]=rsample(train0,K,Kr,ptr); % train is K by M+N
 z{1}=(train(:,1:M))'; % input sample matrix  M by K, layer# = 1
 d=train(:,M+1:MN)';   % corresponding target value  N by K

 % Feed-forward phase, compute sum of square errors
 for l=2:L,              % the l-th layer
    u{l}=w{l}*[ones(1,K);z{l-1}]; % u{l} is n(l) by K
    z{l}=actfun(u{l},atype(l));
 end
 error=d-z{L};           % error is N by K
 E(t)=sum(sum(error.*error));

 % Error back-propagation phase, compute delta error
 delta{L}=actfunp(u{L},atype(L)).*error;  % N (=n(L)) by K
 if L>2,
    for l=L-1:-1:2,
      delta{l}=(w{l+1}(:,2:n(l)+1))'*delta{l+1}...
               .*actfunp(u{l},atype(l));
    end
 end

 % update the weight matrix using gradient,
 % momentum and random perturbation
 for l=2:L,
    dw{l}=alpha*delta{l}*[ones(1,K);z{l-1}]'+...
          mom*dw{l}+randn(size(w{l}))*0.005;
    w{l}=w{l}+dw{l};
 end

 % display the training error
 bpdisplay;     % call mfile bpdisplay.m

 % Test convergence to see if the convergence
 % condition is satisfied,
 cvgtest;       % call mfile cvgtest.m
 t = t + 1;     % increment epoch count
end % while loop
```

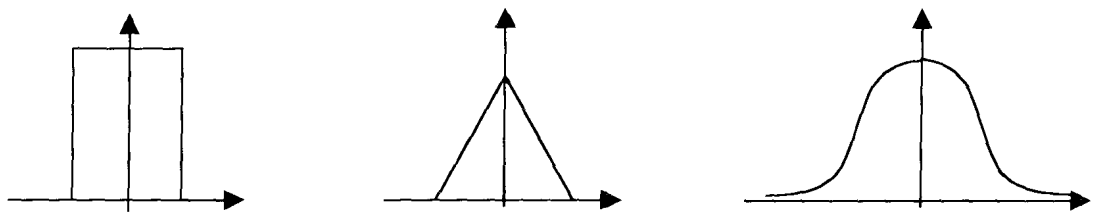
This m-file, called `bp.m`, together with related m-files `bpconfig.m`, `cvgtest.m`, `bpdisplay.m`, and supporting functions, can be downloaded from the CRC website for the convenience of readers.

There are numerous commercial software packages that implement the multilayer perceptron neural network structure. Notably, the MATLAB neural network toolbox™ from Mathwork is a

sophisticated software package. Software packages in C++ programming language that are available free for non-commercial use include PDP++ (http://www.cnbc.cmu.edu/PDP++/PDP++.html) and MLC++ (http://www.sgi.com/tech/mlc/).

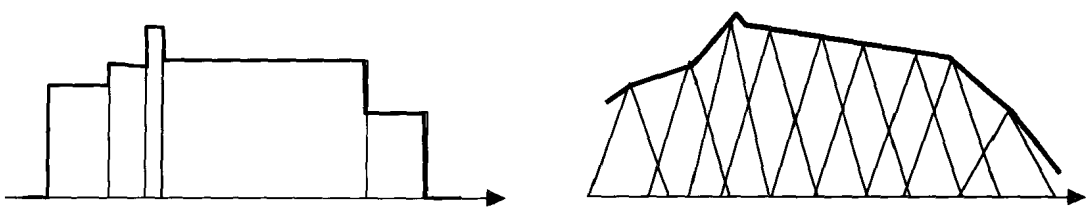
1.2.3 Radial Basis Networks

A radial basis network is a feed-forward neural network using the radial basis activation function. A radial basis function has the general form of $f(||\mathbf{x} - \mathbf{m_0}||) = f(\mathbf{r})$. Such a function is symmetric with respect to a center point $\mathbf{x}_0$. Some examples of radial basis functions in one-dimensional space are depicted in Figure 1.10.



1.10 Three examples of one-dimensional radial basis functions.

Radial basis functions can be used to approximate a given function. For example, as illustrated in Figure 1.11, a rectangular-shaped radial basis function can be used to construct a staircase approximation of a function, and a triangular-shaped radial basis function can be used to construct a trapezoidal approximation of a function.



1.11 Two examples illustrating radial basis function approximation.

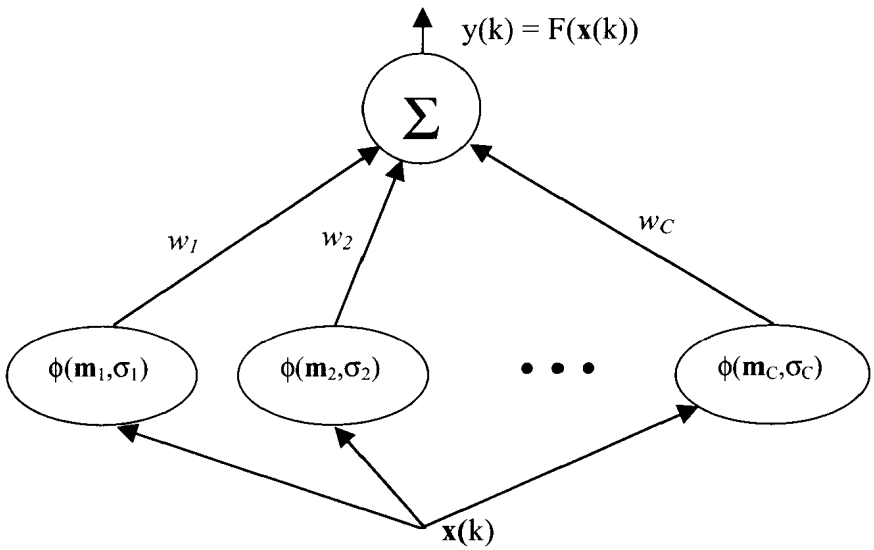
In each of the examples in Figure 1.11, the approximated function can be represented as a weighted linear combination of a family of radial basis functions with different scaling and translations:

$$\hat{F}(x) = \sum_{i=1}^{C} w_i \varphi\left(\|x - m_i\|/\sigma_i\right) . \tag{1.18}$$

This function can be realized with a radial basis network, as shown in Figure 1.12.

There are two types of radial basis networks based on how the radial basis functions are placed and shaped. To introduce these radial basis networks, let us present the function approximation problem formulation:

Radial Basis Function Approximation Problem Given a set of points $\{\mathbf{x}(k);\ 1 \leq k \leq K\}$ and the values of an unknown function $F(\mathbf{x})$ evaluated on these K points $\{d(k) = F(\mathbf{x}(k));\ 1 \leq k \leq K\}$, find an approximation of $F(\mathbf{x})$ in the form of Equation (1.18) such that the sum of square approximation



1.12 A radial basis network.

error at these sets of training samples,

$$\sum_{k=1}^{K} \left[d(k) - \hat{F}(x(k)) \right]^2$$

is minimized.

1.2.3.1 Type I Radial Basis Network

The first type of radial basis network chooses every training sample as the location of a radial basis function [5]. In other words, it sets $C = K$ and $\mathbf{m}_i = \mathbf{x}(i)$, where $1 \leq i \leq K$. Furthermore, a fixed constant scaling parameter σ is chosen for every radial basis function. For convenience, $\sigma = 1$ in the derivation below. That is, $\sigma_i = \sigma$ for $1 \leq k \leq K$. Now rewrite Equation (1.18) in a vector inner product formulation:

$$[\phi(\|x(k) - m_1\|) \quad \phi(\|x(k) - m_2\|) \quad \cdots \quad \phi(\|x(k) - m_C\|)] \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_C \end{bmatrix} = d(k) . \tag{1.19}$$

Substituting $k = 1, 2, \ldots, K$, Equation (1.19) becomes a matrix equation $\mathbf{\Phi w} = \mathbf{d}$:

$$\underbrace{\begin{bmatrix} \phi(\|x(1) - m_1\|) & \phi(\|x(1) - m_2\|) & \cdots & \phi(\|x(1) - m_C\|) \\ \phi(\|x(2) - m_1\|) & \phi(\|x(2) - m_2\|) & \cdots & \phi(\|x(2) - m_C\|) \\ \vdots & \vdots & \ddots & \vdots \\ \phi(\|x(K) - m_1\|) & \phi(\|x(K) - m_2\|) & \cdots & \phi(\|x(K) - m_C\|) \end{bmatrix}}_{\mathbf{\Phi}} \underbrace{\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_C \end{bmatrix}}_{\mathbf{w}} = \underbrace{\begin{bmatrix} d(1) \\ d(2) \\ \vdots \\ d(K) \end{bmatrix}}_{\mathbf{d}} \tag{1.20}$$

$\mathbf{\Phi}$ is a $K \times C$ square matrix (note that $C = K$), and is generally positive for commonly used radial basis functions. Thus, the weight vector $\mathbf{w}$ can be found as:

$$\mathbf{w} = \mathbf{\Phi}^{-1}\mathbf{d} \,. \tag{1.21a}$$

However, in practical applications, the $\mathbf{\Phi}$ matrix may be nearly singular, leading to a numerically unstable solution of $\mathbf{w}$. This can happen when two or more samples $\mathbf{x}(k)$s are too close to each other. Several different approaches can be applied to alleviate this problem.

1.2.3.1.1 Method 1: Regularization

For a small positive number λ, a small diagonal matrix is added to the radial basis coefficient matrix $\mathbf{\Phi}$ such that

$$\mathbf{w} = (\mathbf{\Phi} + \lambda\mathbf{I})^{-1}\mathbf{d} \,. \tag{1.21b}$$

1.2.3.1.2 Method 2: Least Square Using Pseudo-Inverse

The goal is to find a least square solution $\mathbf{w}_{\text{LS}}$ such that $||\mathbf{\Phi w} - \mathbf{d}||^2$ is minimized. Hence,

$$\mathbf{w} = \mathbf{\Phi}^{+}\mathbf{d} \tag{1.22}$$

where $\mathbf{\Phi}^{+}$ is the pseudo-inverse matrix of $\mathbf{\Phi}$ and can be found using singular value decomposition.

1.2.3.2 Type II Radial Basis Network

The type II radial basis network is rooted in the regularization theory [6]. The radial basis function of choice is the Gaussian radial basis function:

$$\phi(\|x - m\|) = \exp\left[-\frac{\|x - m\|^2}{2\sigma^2}\right] .$$

The locations of these Gaussian radial basis function are obtained by clustering the input samples $\{\mathbf{x}(k);\ 1 \le k \le K\}$. Known clustering algorithms such as the k-means clustering algorithm can be applied to serve this purpose. However, there is no objective method to determine the number of clusters. Some experimentation will be needed to find an adequate number of clusters $C(< K)$. Once the cluster is completed, the mean and variance of each cluster can be used as the center location and the spread of the radial basis function. A type II radial basis network gives the solution to the following regularization problem:

Type II Radial Basis Network Approximation Problem Find $\mathbf{w}$ such that $\|\mathbf{Gw}-\mathbf{d}\|^2$ is minimized subject to the constraint $\mathbf{w}^{\mathrm{T}}\mathbf{G}_0\mathbf{w} = \mathbf{a}$ constant.

In the above, $\mathbf{G}$ is a $K \times C$ matrix similar to the $\mathbf{\Phi}$ matrix in Equation (1.20) and is defined as:

$$\mathbf{G} = \begin{bmatrix} \exp\left[-\frac{(x(1)-m_1)^2}{2\sigma_1^2}\right] & \exp\left[-\frac{(x(1)-m_2)^2}{2\sigma_2^2}\right] & \cdots & \exp\left[-\frac{(x(1)-m_C)^2}{2\sigma_C^2}\right] \\ \exp\left[-\frac{(x(2)-m_1)^2}{2\sigma_1^2}\right] & \exp\left[-\frac{(x(2)-m_2)^2}{2\sigma_2^2}\right] & \cdots & \exp\left[-\frac{(x(2)-m_C)^2}{2\sigma_C^2}\right] \\ \vdots & \vdots & \ddots & \vdots \\ \exp\left[-\frac{(x(K)-m_1)^2}{2\sigma_1^2}\right] & \exp\left[-\frac{(x(K)-m_2)^2}{2\sigma_2^2}\right] & \cdots & \exp\left[-\frac{(x(K)-m_C)^2}{2\sigma_C^2}\right] \end{bmatrix}$$

and $\mathbf{G_0}$ is a $C \times C$ symmetric square matrix defined as:

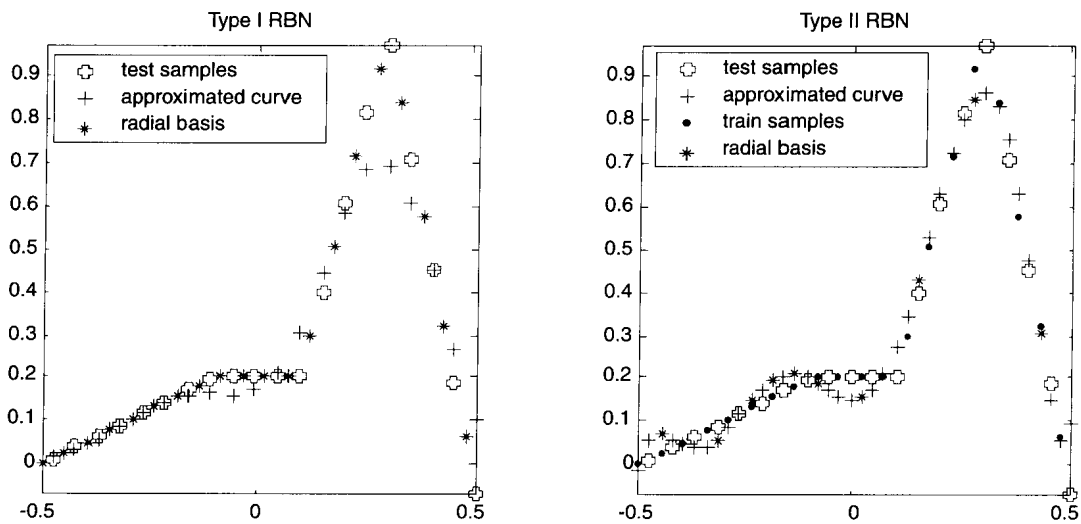
$$\mathbf{G_0} = \begin{bmatrix} \exp\left[-\frac{(m_1-m_1)^2}{2\sigma_1^2}\right] & \exp\left[-\frac{(m_1-m_2)^2}{2\sigma_2^2}\right] & \cdots & \exp\left[-\frac{(m_1-m_C)^2}{2\sigma_C^2}\right] \\ \exp\left[-\frac{(m_2-m_1)^2}{2\sigma_1^2}\right] & \exp\left[-\frac{(m_2-m_2)^2}{2\sigma_2^2}\right] & \cdots & \exp\left[-\frac{(m_2-m_C)^2}{2\sigma_C^2}\right] \\ \vdots & \vdots & \ddots & \vdots \\ \exp\left[-\frac{(m_C-m_1)^2}{2\sigma_1^2}\right] & \exp\left[-\frac{(m_C-m_2)^2}{2\sigma_2^2}\right] & \cdots & \exp\left[-\frac{(m_C-m_C)^2}{2\sigma_C^2}\right] \end{bmatrix}$$

The solution to the constrained optimization problem can be found as:

$$\mathbf{w} = \left(\mathbf{G}^{\mathrm{T}}\mathbf{G} + \lambda\mathbf{G_0}\right)^{-1}\mathbf{G}^{\mathrm{T}}\mathbf{d} \tag{1.23}$$

where λ is a regularization parameter and is usually selected as a very small non-negative number. As $\lambda \to 0$, Equation (1.23) becomes the least square solution.

MATLAB Implementation The type I and type II radial basis networks have been implemented in a MATLAB m-file called `rbn.m`. Using this function, we developed a demonstration program called `rbndemo.m` that can illustrate the properties of these two types of radial basis networks. Twenty training samples are regularly spaced in $[-0.5\ 0.5]$, and the function to be approximated is a piecewise linear function. For the type I RBN network, 20 Gaussian basis functions located at each training sample are used. The standard deviation of each Gaussian basis function is the same and equals the average distance between two basis functions. For the type II RBN network, ten Gaussian basis functions are generated using k-means clustering algorithm. The variance of each Gaussian basis function is the variance of samples within the corresponding cluster.



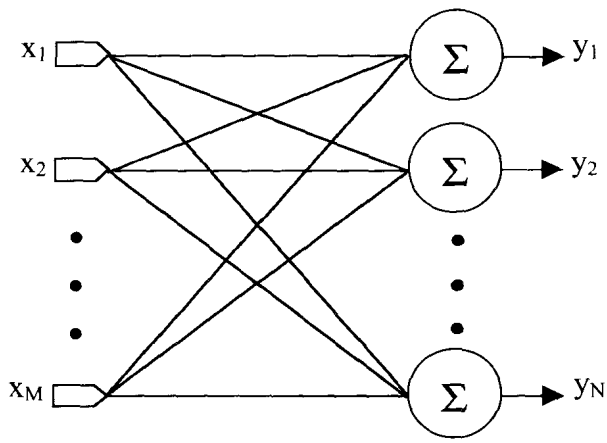
1.13 Simulation results demonstrating type I and type II RBN networks.

1.2.4 Competitive Learning Networks

Both the multilayer perceptron and the radial basis network are based on the popular learning paradigm of error-correction learning. The synaptic weights of these networks are adjusted to reduce the difference (error) between the desired target value and corresponding output. For competitive learning networks, a competitive learning paradigm is incorporated.

With the competitive learning paradigm, a single-layer of neurons compete among themselves to represent the current input vector. The winning neuron will adjust its own weight to be closer to the input pattern. As such, competitive learning can be regarded as a sequential clustering algorithm.

1.2.4.1 Orthogonal Linear Networks



1.14 An orthogonal linear network.

In a single-layer, linear network, the output $y_n(t) = \Sigma_{m=1}^{M} w_{nm}(t)x_m(t)$. The synaptic weights are updated according to a generalized Hebbian learning rule [7]:

$$\Delta w_{nm}(t) = w_{nm}(t+1) - w_{nm}(t) = \eta y_n(t)\left[x_m(t) - \sum_{k=1}^{n} w_{km}(t)y_k(t)\right] .$$

As such, the weight vector $\mathbf{w}_n = [w_{n1}\ w_{n2} \ldots w_{nM}]^T$ will converge to the eigenvector of the nth largest eigenvalue of the sample covariance matrix formed by the input vectors

$$\mathbf{C} = \sum_{t} \mathbf{x}(t)\mathbf{x}^T(t)$$

where

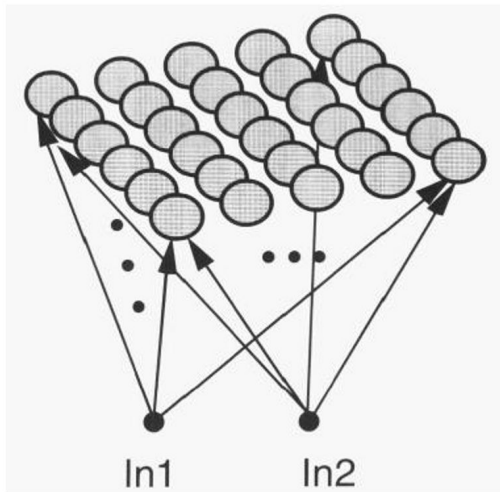
$$\mathbf{x}^T(t) = [x_1(t)\, x_2(t)\ \ldots x_M(t)] .$$

Therefore, upon convergence, such a generalized Hebbian learning network will produce the principal components (eigenvectors) of the sample covariance matrix of the input samples. Principal component analysis (PCA) has found numerous applications in data compression and data analysis tasks. For the signal processing applications, PCA based on an orthogonal linear network has been applied to image compression [7].

A MATLAB implementation of the generalized Hebbian learning algorithm and its demonstration can be found in `ghademo.m` and `ghafun.m`.

1.2.4.2 Self-Organizing Maps

A self-organizing map [8] is a single-layer, competitive neural network that imposes a preassigned ordering among the neurons. For example, in Figure 1.15, the shaded circles represent a 6×5 array of neurons, each labeled with a preassigned index (i, j), $1 \leq i \leq 6, 1 \leq j \leq 5$. In1



1.15 A two-dimensional self-organizing map neural network structure.

and In2 are two-dimensional inputs. Given a specific neuron, e.g., (3,2), one may identify its four nearest neighbors as (3,1), (2,2), (4,2), and (3,3). Each neuron has two synaptic connections to the two inputs. The weights of these two connections give a two-dimensional coordinate to represent the location of the neuron in the input feature space. If the input (In1, In2) is very close to the two weights of a neuron, that neuron will give an output 1, signifying it is the winner to represent the current input feature vector. The remaining losing neurons will have their output remain at 0. Therefore, the self-organizing map is a neural network whose behavior is governed by competitive learning. In the ideal situation, each neuron will represent a cluster of input feature vectors (points) that may share some common semantic meaning. Consequently, the 6×5 array of neurons can be regarded as a mapping from points in the input feature space to a coarsely partitioned label space through the process of clustering. The initial labeling of individual neurons allows features of similar semantic meaning to be grouped into closer clusters. In this sense, the self-organizing map provides an efficient method to visualize high-dimensional data samples in low-dimensional display.

1.2.4.2.1 Basic Formulation of Self-Organizing Maps (SOMs)

Initialization: Choose weight vectors $\{w_m(0); 1 \leq m \leq M\}$ randomly. Set iteration count $t = 0$.
While Not_Converged
  Choose the next x and compute $d(x, w_m(t))$; $1 \leq m \leq M$.
  Select $m^* = \text{mim}_m d(x, w_m(t))$

$$w_m(t+1) = \begin{cases} w_m(t) + \eta(x - w_m(t)) & m \in N(m^*, t); \\ w_m(t) & m \notin N(m^*, t) \end{cases}$$

  % Update node m^* and its neighborhood nodes:
  If Not_converged, then $t = t + 1$
End % while loop

This algorithm is demonstrated in a MATLAB program `somdemo.m`. A plot is given in Fig-