Table 11.1. *Transitional behavior per state in the two-neuron Hopfield net. Act_i denotes the activation of neuron i.*

Current states	Energy levels	V_1	V_2	Chosen neuron	Act_1	Act_2		V_1	V_2	Next state
A	0	0	0	1	−0.7	—	⇒	0	0	A
				2	—	0.1	⇒	0	1	B
B	-0.1	0	1	1	−0.4	—	⇒	0	1	B
				2	—	0.1	⇒	0	1	B
C	0.7	1	0	1	−0.7	—	⇒	0	0	A
				2	—	0.4	⇒	1	1	D
D	0.3	1	1	1	−0.4	—	⇒	0	1	B
				2	—	0.4	⇒	1	1	D

Skiing Down the Energy Slope toward a Stable Well

This subsection presents the operational picture of the binary Hopfield net to understand more thoroughly "*state transitions*" with a fixed set of weights. The network can be seen as a dynamical system moving through a sequence of states toward a *stable state* over time. For computational simplicity, we describe a small two-neuron binary Hopfield net illustrated in Figure 11.13(a). We consider a particular weight parameter setup as follows:

$$T_{12} = T_{21} = 0.3,\ U_1 = 0.7,\ U_2 = 0.1.$$

Recall that each neuron can have one of two states. Hence, this two-neuron binary network can have a total of four states: state A, state B, state C, and state D, corresponding to the four pairs of (V_1, V_2): $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$, respectively. According to Equation (11.17),

$$E = -T_{12}V_1V_2 + V_1U_1 + V_2U_2 = -0.3V_1V_2 + 0.7V_1 + 0.1V_2.$$

Using the preceding equation, we can calculate the energy levels of the four states: $E = 0$ for state A, $E = -0.1$ for state B, $E = 0.7$ for state C, and $E = 0.3$ for state D.

In light of Equation (11.16), we consider two activations:

$$Act_1 \equiv T_{12}V_2 - U_1 = 0.3V_2 - 0.7,$$

$$Act_2 \equiv T_{21}V_1 - U_2 = 0.3V_1 + 0.1.$$

For state A ($V_1 = V_2 = 0$), Act_1 at neuron 1 is negative (-0.7), and Act_2 at neuron 2 is positive (0.1). Therefore, if neuron 2 starts firing, state A transits to state B ($V_1 = 0$, $V_2 = 1$). If neuron 1 starts firing, however, state A does not transit. Table 11.1 summarizes all state transitions and energy levels.

In terms of storage capacity, the number of memories is estimated to be nearly 10% to 20% of the number of neurons in the Hopfield net [12]. Although the Hopfield memory net is not very efficient, its mechanism based on the *energy* concept is worth exploring. The basic binary Hopfield net is first described.

11.7.2 Binary Hopfield Networks

Formulation

Each interconnection has a weight (or connection strength), denoted by T_{ij} from neuron j to neuron i. The Hopfield network considers bidirectionality in the connections, using the symmetric weight matrix, $T_{ij} = T_{ji}$, and also assumes that no neuron is connected to itself ($T_{ii} = 0$). In Hopfield's early analysis, each neuron has a *binary state* of either 0 or 1; those neurons subsequently form the *binary* Hopfield net. At each moment, the entire state of the network can be represented by a binary state vector. The neurons are assumed to be threshold logic units so that a state has one of the two possible values. The following is the *firing rule* of an arbitrary neuron i:

$$V_i = \begin{cases} 1 & \text{if } \sum_{j \neq i} T_{ij} V_j > U_i, \\ 0 & \text{if } \sum_{j \neq i} T_{ij} V_j < U_i. \end{cases}$$

where V_i denotes the output of neuron i and U_i its threshold. This can be rewritten with a neuron function, $f(\cdot)$:

$$\begin{aligned} &V_i = f(\textstyle\sum_{j=1, j \neq i}^{n} T_{ij} V_j - U_i), \\ &\text{where } f(x) = \operatorname{sgn}(x) \equiv \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{if } x < 0. \end{cases} \end{aligned} \tag{11.16}$$

Each network state has an associated *energy*[2] in a quadratic form:

$$E = -\frac{1}{2} \sum \sum_{j \neq i} T_{ij} V_j V_i + \sum V_i U_i. \tag{11.17}$$

The change in energy (ΔE) with respect to the change of state at neuron i (ΔV_i) is derived from Equation (11.17):

$$\Delta E = -\Delta V_i (\sum_{j \neq i} T_{ij} V_j - U_i). \tag{11.18}$$

When a neuron i alters its state according to the firing rule defined in Equation (11.16), it is thus ensured that ΔE is always negative. In other words, E is a monotonically decreasing function of the network state.

[2]The energy in Equation (11.17) has a quadratic form similar to the kinetic energy E_k of a mass m at velocity v:

$$E_k = \frac{1}{2} m v^2.$$

In many cases, *physical energy* can be represented in such a quadratic form.

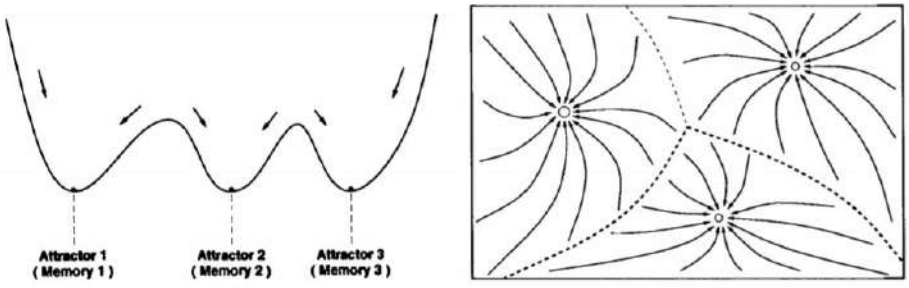


Figure 11.12. *Images of 1-D and 2-D energy terrains configured by three attractors in a Hopfield network. The dots denote stable states where patterns are memorized. The network state moves in the direction of the arrows to one of the attractors, which is determined by the starting point (i.e., the given input pattern).*

> *can therefore be regarded as a general content-addressable memory. The physical system will be a potentially useful memory if, in addition, any prescribed set of states can readily be made the stable states of the system.*

The Hopfield net highlights a content-addressable memory and a tool for solving an optimization problem. These features are discussed next.

11.7.1 Content-Addressable Nature

The Hopfield net normally develops a number of locally stable points in state space. Because the network's dynamics minimize *energy*, other points in state space drain into the stable points (called **attractors** or **wells**), which are (possibly local) energy minima.

The Hopfield net realizes the operation of a *content-addressable (auto-associative) memory* in the sense that newly presented input patterns (or arbitrary initial states) can be connected to the appropriate patterns stored in memories (i.e., attractors or stable states). The presented input pattern vector cannot escape from a region, what we call a **basin of attraction**, configured by each attractor (Figure 11.12). Restated, the network produces a desired memorized pattern in response to the given pattern. The initial network state is assumed to lie within the reach of the fixed attractors' *basins of attraction*. That is, the entire configuration space is divided into different basins of attraction. Notably, an attractor is regarded as a fixed point if it is unique in state space. In general, however, an attractor may have *chaos* or the *limit cycle* of a periodic sequence of states. An advantage of neural networks (NNs) lies in their *fault tolerance*—more specifically, in that NNs are tolerant of a presented pattern's slight distortions. This NN feature is appropriate for performing the primary task of content-addressable memory.

total least-squares (TLS) method [15] for data fitting; the first principal component direction [5, 29, 47] can be used to achieve the same goal. In comparison, the standard least-squares (LS) method, as described in Chapter 5, attempts to find a least-squares regression line that minimizes the vertical distance ($\sum_i li$) from the data points to the line, as indicated in Figure 11.11.

11.6.2 Oja's Modified Hebbian Rule

By using the simple neural network in Figure 11.9, Oja demonstrated that with a Hebbian-type learning rule, the network performs PCA [42].

Employing the plain Hebbian learning rule in Equation (11.10) leads to unlimited growth of the weight vector. One solution is to renormalize the weight vector after each update:

$$w_i(t+1) = \frac{w_i(t) + \eta\, y(t)\, x_i(t)}{\sqrt{\sum_{i=1}^{n} [w_i(t) + \eta\, y(t)\, x_i(t)]^2}}. \tag{11.14}$$

If the learning rate η is small, the preceding equation at $\eta = 0$ can be expanded using Taylor series expansion. Deleting the second- and higher-order terms yields

$$\Delta w_i = \eta y x_i - \eta y^2 w_i = \eta y (x_i - y w_i). \tag{11.15}$$

The preceding equation is the **modified Hebbian learning rule**, which entails adding a *weight decay* proportional to the squared output (y^2) to maintain the weight vector unit length automatically. This Hebbian-type adaptation involves less computation since the normalization operation in Equation (11.14) is not required. Hertz et al. pointed out that Equation (11.15) resembles reverse LMS learning (Table 11.4 in Section 11.8), in which the weight updating is based on the difference between the actual input and the backpropagated output [23].

Oja also indicated that the weight vector $\mathbf{w}$ approaches an eigenvector of the correlation matrix $\mathbf{R}$ with the largest eigenvalue. The larger the eigenvalue, the more precise the direction of the corresponding principal component (or eigenvector). Oja later extended the formulation for multiple-output systems to perform PCA [43].

11.7 THE HOPFIELD NETWORK

In 1982, Hopfield proposed the so-called **Hopfield network**, which possesses *auto-associative* properties. It is a **recurrent** (or *fully interconnected*) network in which all neurons are connected to each other, with the exception that no neuron has any connection to itself. In the network configuration, he embodied the *physical principle*, and set up an **energy function**. The concept derives from a physical system [25]:

> *Any physical system whose dynamics in phase space is dominated by a substantial number of locally stable states to which it is attracted*

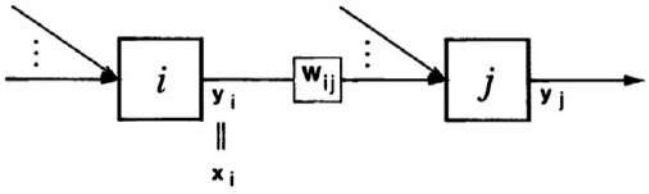


Figure 11.8. *A simple network topology for Hebbian learning; weight w_{ij} resides between two neurons i and j.*

Step 5: If the maximum number of iterations is reached, stop. Otherwise, return to step 3.

Two improved versions of LVQ are available; both of them attempt to use the training data more efficiently by updating the winner and the runner-up (the next closest vector) under a certain condition. The improved versions are called LVQ2 and LVQ3; refs. [34] and [35] provide further details.

11.5 HEBBIAN LEARNING

Hebb [20] described a simple learning method of synaptic weight change. When two cells fire simultaneously (i.e., have strong responses), their connection strength (or weight) increases. Such phenomenon is the so-called **Hebbian learning**, where the weight increase between two neurons is proportional to the frequency at which they fire together. Among various mathematical formulas of this principle, the simplest one is expressed as

$$\Delta w_{ij} = \eta y_i y_j, \tag{11.6}$$

where η is the learning rate. Since weights are adjusted according to the correlation of neuron outputs, the preceding formula is a type of *correlational learning rule*. The ith neuron's output y_i can be regarded as an input (x_i) to another neuron j (Figure 11.8), so Equation (11.6) can be written as

$$\Delta w_{ij} = \eta y_j x_i. \tag{11.7}$$

Restated, a weight is assumed to change proportionately to the *correlation* of the input and output signals. By using a neuron function $f(\cdot)$, y_j is given by

$$y_j = f(\mathbf{w}_j^T \mathbf{x}).$$

Thus, Equation (11.7) is equivalent to the following:

$$\Delta w_{ij} = \eta\, f(\mathbf{w}_j^T \mathbf{x})\, x_i. \tag{11.8}$$

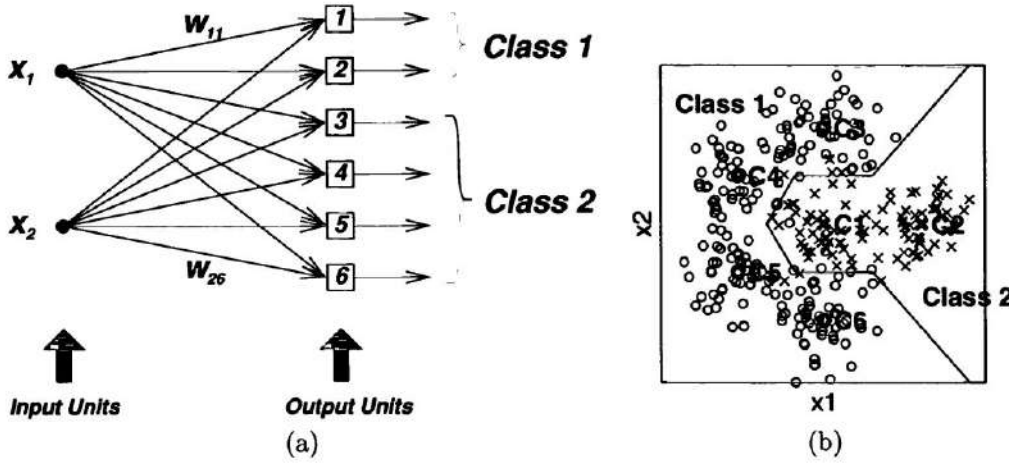


Figure 11.7. *Learning vector quantization (LVQ): (a) network representation; (b) possible data distribution and decision boundary.* [MATLAB command for (b): `lvqdata`]

vector $\mathbf{x}$ must be found. If $\mathbf{x}$ and $\mathbf{w}$ belong to the same class, we move $\mathbf{w}$ toward $\mathbf{x}$; otherwise we move $\mathbf{w}$ away from the input vector $\mathbf{x}$.

After learning, an LVQ network classifies an input vector by assigning it to the same class as the output unit that has the weight vector (cluster center) closest to the input vector. Figure 11.7(b) illustrates a possible distribution of data set and weights after training.

A sequential description of the LVQ method is as follows:

Step 1: Initialize the cluster centers by a clustering method.

Step 2: Label each cluster by the voting method.

Step 3: Randomly select a training input vector $\mathbf{x}$ and find k such that $\|\mathbf{x} - \mathbf{w}_k\|$ is a minimum.

Step 4: If $\mathbf{x}$ and $\mathbf{w}_k$ belong to the same class, update $\mathbf{w}_k$ by

$$\Delta \mathbf{w}_k = \eta(\mathbf{x} - \mathbf{w}_k).$$

Otherwise, update $\mathbf{w}_k$ by

$$\Delta \mathbf{w}_k = -\eta(\mathbf{x} - \mathbf{w}_k).$$

The learning rate η is a positive small constant and should decrease with each iteration.

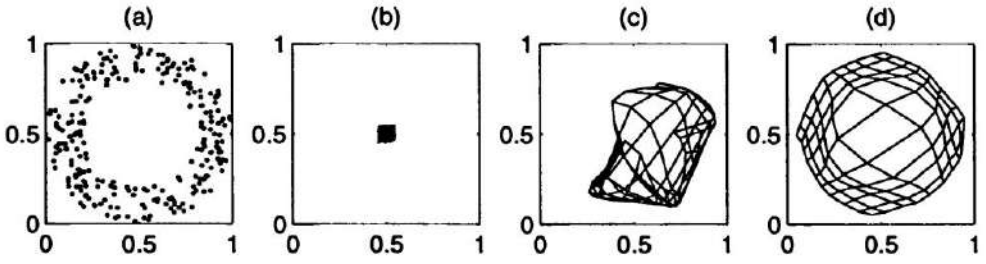


Figure 11.6. *Simulation of the Kohonen self-organizing network: (a) input data uniformly distributed within a doughnut-shaped region; (b) initial weights; (c) weights after 30 iterations; (d) weights after 1000 iterations.* (MATLAB command: `kfm(5)`)

11.4 LEARNING VECTOR QUANTIZATION

Learning vector quantization (LVQ) [33, 34, 35] is an adaptive data classification method based on training data with desired class information. Although a *supervised* training method, LVQ employs unsupervised data-clustering techniques (e.g., competitive learning, introduced in Section 11.2) to preprocess the data set and obtain cluster centers.

LVQ's network architecture closely resembles that of a competitive learning network, except that each output unit is associated with a class. Figure 11.7(a) presents an example, where the input dimension is 2 and the input space is divided into six clusters. The first two clusters belong to class 1, while the other four clusters belong to class 2. The LVQ learning algorithm involves two steps. In the first step, an unsupervised learning data clustering method is used to locate several cluster centers without using the class information. In the second step, the class information is used to fine-tune the cluster centers to minimize the number of misclassified cases.

During the first step of unsupervised learning, any of the data clustering techniques introduced in this chapter and Chapter 15 can be used to identify cluster centers (or weight vectors leading to output units) to represent the data set with no class information. The number of clusters can either be specified *a priori* or determined via a cluster technique capable of adaptively adding new clusters when necessary. Once the clusters are obtained, their classes must be labeled before moving to the second step of supervised learning. Such labeling is achieved by the so-called *voting method* (i.e., a cluster is labeled class k if it has data points belonging to class k as a majority within the cluster.) The clustering process for LVQ is based on the general assumption that similar input patterns generally belong to the same class.

During the second step of supervised learning, the cluster centers are fine-tuned to approximate the desired decision hypersurface. The learning method is straightforward. First, the weight vector (or cluster center) $\mathbf{w}$ that is closest to the input

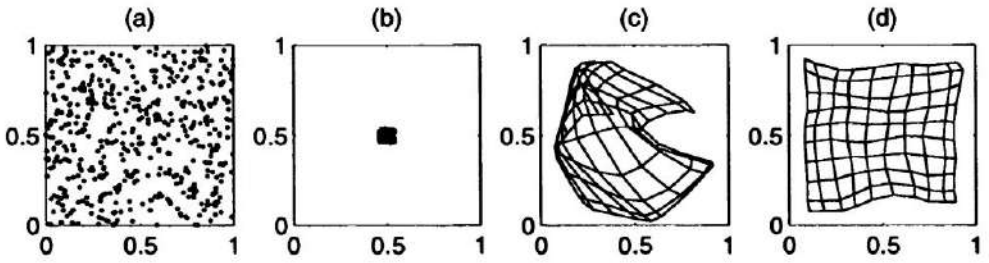


Figure 11.4. *Simulation of the Kohonen self-organizing network: (a) input data uniformly distributed within* $[0,1] \times [0.1]$*; (b) initial weights; (c) weights after 30 iterations; (d) weights after 1000 iterations.* (MATLAB command: `kfm(1)`)

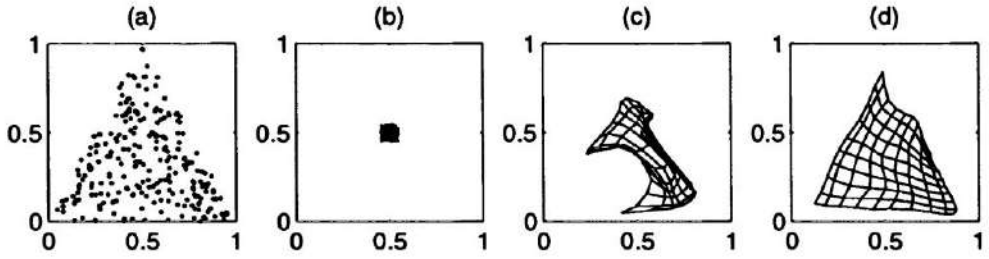


Figure 11.5. *Simulation of the Kohonen self-organizing network: (a) input data uniformly distributed within a triangular region; (b) initial weights; (c) weights after 30 iterations; (d) weights after 1000 iterations.* (MATLAB command: `kfm(2)`)

where p_i and p_c are the positions of the output units i and c, respectively, and σ reflects the scope of the neighborhood. By using the neighborhood function, the update formula can be rewritten as

$$\Delta \mathbf{w}_i = \eta \Omega_c(i)(\mathbf{x} - \mathbf{w}_i), \text{ where } i \text{ is the index for all output units.}$$

To achieve a better convergence, the learning rate η and the size of neighborhood (or σ) should be decreased gradually with each iteration. Figures 11.4, 11.5, and 11.6 present simulation results of Kohonen feature maps with different input data distributions; the output units are arranged in a 10-by-10 two-dimensional mesh. In the simulation, η and σ linearly decreased with the number of iterations.

The most well-known application of Kohonen self-organizing networks is Kohonen's attempt to construct a neural phonetic typewriter [32] that is capable of transcribing speech into written text from an unlimited vocabulary, with an accuracy of 92% to 97%. The network has also been used to learn ballistic arm movements [44].

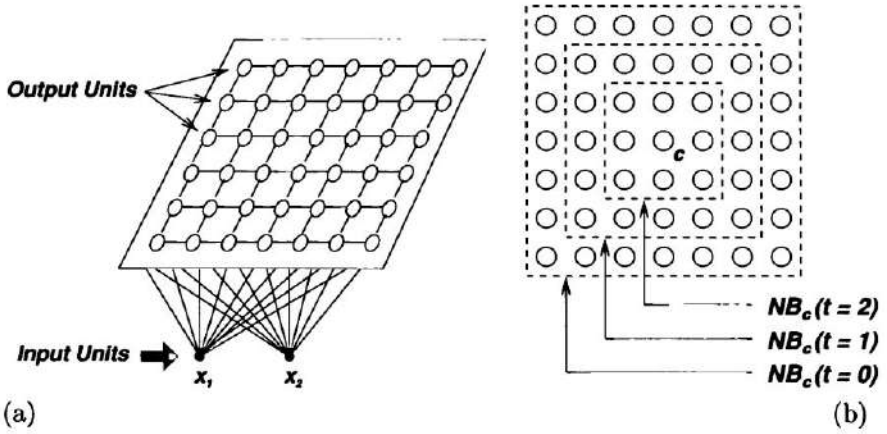


Figure 11.3. *(a) A Kohonen self-organizing network with 2 input and 49 output units; (b) the size of a neighborhood around a winning unit decreases gradually with each iteration.*

winning unit's weights but also all of the weights in a neighborhood around the winning units. The neighborhood's size generally decreases slowly with each iteration, as indicated in Figure 11.3(b). A sequential description of how to train a Kohonen self-organizing network is as follows:

Step1: Select the winning output unit as the one with the largest similarity measure (or smallest dissimilarity measure) between all weight vectors $\mathbf{w}_i$ and the input vector $\mathbf{x}$. If the Euclidean distance is chosen as the dissimilarity measure, then the winning unit c satisfies the following equation:

$$\|\mathbf{x} - \mathbf{w}_c\| = \min_i \|\mathbf{x} - \mathbf{w}_i\|,$$

where the index c refers to the winning unit.

Step2: Let NB_c denote a set of index corresponding to a neighborhood around winner c. The weights of the winner and its neighboring units are then updated by

$$\Delta \mathbf{w}_i = \eta(\mathbf{x} - \mathbf{w}_i), \ i \in NB_c,$$

where η is a small positive learning rate. Instead of defining the neighborhood of a winning unit, we can use a **neighborhood function** $\Omega_c(i)$ around a winning unit c. For instance, the Gaussian function can be used as the neighborhood function:

$$\Omega_c(i) = \exp\left(\frac{-\|p_i - p_c\|^2}{2\sigma^2}\right),$$

Chapter 7. Therefore, one of the following formulas for η is commonly used:

$$\begin{cases} \eta(t) = \eta_0 e^{-\alpha t}, \text{ with } \alpha > 0, \text{ or} \\ \eta(t) = \eta_0 t^{-\alpha}, \text{ with } \alpha \leq 1, \text{ or} \\ \eta(t) = \eta_0(1 - \alpha t), \text{ with } 0 < \alpha < (\max\{t\})^{-1}. \end{cases}$$

Competitive learning lacks the capability to add new clusters when deemed necessary. Moreover, if the learning rate η is a constant, competitive learning does not guarantee stability in forming clusters; the winning unit that responds to a particular pattern may continue changing during training. On the other hand, η, if decreasing with time, may become too small to update cluster centers when new data of a different probability nature are presented. Carpenter and Grossberg referred to such an occurrence as the **stability-plasticity dilemma**, which is common in designing intelligent learning systems [8]. In general, a learning agent (or system) should be *plastic*, or adaptive in reacting to changing environments; meanwhile, it should be *stable* to preserve knowledge acquired previously. **Adaptive resonance theory** (**ART**), as introduced by Grossberg [17], proposes a solution to this dilemma. Based on ART, Carpenter and Grossberg proposed a series of similar networks, including ART1, ART2 [7], ART3 [9], and ARTMAP [10].

If the output units of a competitive learning network are arranged in a geometric manner (such as in a one-dimensional vector or two-dimensional array), then we can update the weights of the winners as well as the *neighboring* losers. Such a capability corresponds to the notion of Kohonen feature maps, as discussed in the next section.

After competitive learning is finished, the input space is divided into a number of disjoint clusters, each of which is represented by a cluster center. These cluster centers are also known as *template*, *reference vector*, or *codebook vector* [16, 38]. For an input vector, we can use the corresponding template to represent the input vector rather than the vector itself. Such an approach is called **vector quantization** and it has been used for data compression in image processing and communication systems. Section 11.4 introduces a *supervised* version of vector quantization, known as learning vector quantization [33, 34, 35]. Other competitive learning applications include graph bipartitioning [22, 45] and word perception models [45].

11.3 KOHONEN SELF-ORGANIZING NETWORKS

Kohonen self-organizing networks [30, 31], also known as **Kohonen feature maps** or **topology-preserving maps**, are another competition-based network paradigm for data clustering. Networks of this type impose a neighborhood constraint on the output units, such that a certain topological property in the input data is reflected in the output units' weights.

Figure 11.3(a) presents a relatively simple Kohonen self-organizing network with 2 inputs and 49 outputs. The learning procedure of Kohonen feature maps is similar to that of competitive learning networks. That is, a similarity (dissimilarity) measure is selected and the winning unit is considered to be the one with the largest (smallest) activation. For Kohonen feature maps, however, we update not only the

$$
\begin{aligned}
d_1 &= c_1 \exp\left[-\frac{\|\mathbf{x}_1 - \mathbf{x}_1\|^2}{2\sigma_1^2}\right] + \cdots + c_n \exp\left[-\frac{\|\mathbf{x}_1 - \mathbf{x}_n\|^2}{2\sigma_n^2}\right], \\
d_2 &= c_1 \exp\left[-\frac{\|\mathbf{x}_2 - \mathbf{x}_1\|^2}{2\sigma_1^2}\right] + \cdots + c_n \exp\left[-\frac{\|\mathbf{x}_2 - \mathbf{x}_n\|^2}{2\sigma_n^2}\right], \\
&\vdots \\
d_n &= c_1 \exp\left[-\frac{\|\mathbf{x}_n - \mathbf{x}_1\|^2}{2\sigma_1^2}\right] + \cdots + c_n \exp\left[-\frac{\|\mathbf{x}_n - \mathbf{x}_n\|^2}{2\sigma_n^2}\right].
\end{aligned}
$$

Writing them in matrix form, we obtain

$$
\begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} \exp\left[-\frac{\|\mathbf{x}_1-\mathbf{x}_1\|^2}{2\sigma_1^2}\right] & \cdots & \exp\left[-\frac{\|\mathbf{x}_1-\mathbf{x}_n\|^2}{2\sigma_n^2}\right] \\ \exp\left[-\frac{\|\mathbf{x}_2-\mathbf{x}_1\|^2}{2\sigma_1^2}\right] & \cdots & \exp\left[-\frac{\|\mathbf{x}_2-\mathbf{x}_n\|^2}{2\sigma_n^2}\right] \\ \vdots & \vdots & \vdots \\ \exp\left[-\frac{\|\mathbf{x}_n-\mathbf{x}_1\|^2}{2\sigma_1^2}\right] & \cdots & \exp\left[-\frac{\|\mathbf{x}_n-\mathbf{x}_n\|^2}{2\sigma_n^2}\right] \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}. \tag{9.22}
$$

Rewriting the preceding in a compact form, we have

$$
\mathbf{D} = \mathbf{G}\mathbf{C}, \tag{9.23}
$$

where

$$
\mathbf{D} = [d_1, d_2, \ldots, d_n]^T, \quad \mathbf{C} = [c_1, c_2, \ldots, c_n]^T.
$$

When the matrix $\mathbf{G}$ is nonsingular, we have a unique solution:

$$
\mathbf{C} = \mathbf{G}^{-1}\mathbf{D}, \tag{9.24}
$$

where $\mathbf{G}^{-1}$ denotes the inverse matrix of $\mathbf{G}$. In practice, however, $\mathbf{G}$ may be ill-conditioned (close to singularity), especially when the training set is large. The regularization approach [12, 41] allows for such cases by modifying $\mathbf{G}$ to $\mathbf{G} + \lambda\mathbf{I}$, where λ is a positive real number (regularization parameter) and $\mathbf{I}$ is the identity matrix. (Similar modifications can be found in the Levenberg-Marquardt method to make the Hessian matrix positive definite; see Chapter 6 or refer to refs. [28, 30, 50].) Poggio and Girosi [41] took this approach in implementing a regularization network. Specht [56] presents a general regression neural network (GRNN) using a (nonlinearly) weighted average of the given training samples; the GRNN can be described by a topology identical to the RBFN with weighted average in Figure 9.10(b).

When there are fewer basis functions than there are available training samples, an initial guess is required to determine their center positions. We then have an **approximation RBFN**. In this case, matrix $\mathbf{G}$ is not square and the least-squares methods described in Chapter 5 are commonly used to find the matrix $\mathbf{C}$ in Equation (9.23). Note that the matrix $\mathbf{G}$ may be ill-conditioned and limited precision

The conditions under which an RBFN and a FIS are functionally equivalent are summarized as follows [18].

- Both the RBFN and the FIS under consideration use the same aggregation method (namely, either weighted average or weighted sum) to derive their overall outputs.
- The number of receptive field units in the RBFN is equal to the number of fuzzy if-then rules in the FIS.
- Each radial basis function of the RBFN is equal to a multidimensional composite MF of the premise part of a fuzzy rule in the FIS. One way to achieve this is to use Gaussian MFs with the same variance in a fuzzy rule, and apply product to calculate the firing strength. The multiplication of these Gaussian MFs becomes a multidimensional Gaussian function—a radial basis function in RBFN. (See Figure 13.3 for more details.)
- Corresponding radial basis function and fuzzy rule should have the same response function. That is, they should have the same constant terms (for the original RBFN and zero-order Sugeno FIS) or linear equations (for the extended RBFN and first-order Sugeno FIS)..

The functional equivalence between FIS and RBFN cross-fertilizes both computing paradigms. Further details are presented in Section 12.4.

9.5.3 Interpolation and Approximation RBFNs

Assuming that there is no noise in the training data set, we need to estimate a function $d(\cdot)$ that yields exact desired outputs for all training data. This task is usually called an "interpolation" problem, and the resultant function $d(\cdot)$ should pass through all of the training data points. When we use an RBFN with the same number of basis functions as we have training patterns, we have a so-called **interpolation RBFN**, where each neuron in the hidden layer reponds to one particular training input pattern.

Consider a Gaussian basis function centered at u_i with a width parameter σ:

$$w_i = R_i(\|\mathbf{x} - \mathbf{u}_i\|) = \exp\left[-\frac{(\mathbf{x} - \mathbf{u}_i)^2}{2\sigma_i^2}\right]. \tag{9.20}$$

Each training input x_i serves as a center for for the basis function, R_i. Thus, from Equation (9.16), we have a Gaussian interpolation RBFN:

$$d(\mathbf{x}) = \sum_{i=1}^{n} c_i \exp\left[-\frac{(\mathbf{x} - \mathbf{x}_i)^2}{2\sigma_i^2}\right]. \tag{9.21}$$

For given σ_i, $i = 1, \ldots, n$, we obtain the following n simultaneous linear Equations for the n unknown weight coefficients, c_i:

rithms for RBFNs have been reported. The receptive field functions are first fixed, and then the weights of the output layer are adjusted. Several schemes have been proposed to determine the center positions ($\mathbf{u}_i$) of the receptive field functions. Lowe [26] proposed a way to determine the centers based on standard deviations of training data. Moody and Darken [34, 35] selected the centers $\mathbf{u}_i$ by means of data clustering techniques (see Chapter 15) that assume that similar input vectors produce similar outputs; σ_i's are then obtained heuristically by taking the average distance to the several nearest neighbors of u_i's. In another variation, Nowlan [38] employed the so-called *soft competition* among Gaussian hidden units to locate the centers. This soft competition method is based on a "maximum likelihood estimate" for the centers, in contrast to the so-called *hard* competitions such as the k-means winner-take-all algorithm.

Once these nonlinear parameters are fixed and the receptive fields are frozen, the linear parameters (i.e., the weights of the output layer) can be updated using either the least-squares method or the gradient method. Alternatively, we can apply the pseudoinverse method in solving Equation (9.23) to determine these weights [5]. Chen et al. [6] used another method that employs the orthogonal least-squares algorithm to determine the $\mathbf{u}_i$'s and c_i's while keeping the σ_i's at predetermined values. There are many other schemes as well, such as generalization properties [2], and sequential adaptation [22], among others [19, 36].

9.5.2 Functional Equivalence to FIS

An extension of the originally proposed Moody-Darken's RBFN is to assign a linear function as the output function of each receptive field; that is, C_i is a linear function of the input variables instead of a constant:

$$C_i = \vec{a}_i \cdot \vec{x} + b_i, \tag{9.19}$$

where $\vec{a}_i$ is a parameter vector and b_i is a scalar parameter. Stokbro et al. [57] used this structure to model the Mackey-Glass chaotic time series [27] and found that this extended version performed better than the originally proposed RBFN with the same number of fitting parameters. Using Equation (9.19), the extended RBFN response given by Equation (9.16) or Equation (9.17) is identical to the response produced by the first-order Sugeno fuzzy inference system (FIS) discussed in Chapter 4, provided that the membership functions, the radial basis functions, and certain operators are choose correctly.

While the RBFN consists of radial basis functions, the FIS comprises a certain number of membership functions. With those radially shaped functions, both FIS and RBFN have a mechanism whereby they can produce a center-weighted response to small receptive fields, localizing the primary input excitation. Although the FIS and the RBFN were developed on different bases, they are essentially rooted in the same soil. Just as the RBFN enjoys quick convergence, the FIS can evolve to recognize some feature in a training data set quickly compared with simple back-propagation MLPs (see Chapter 12).

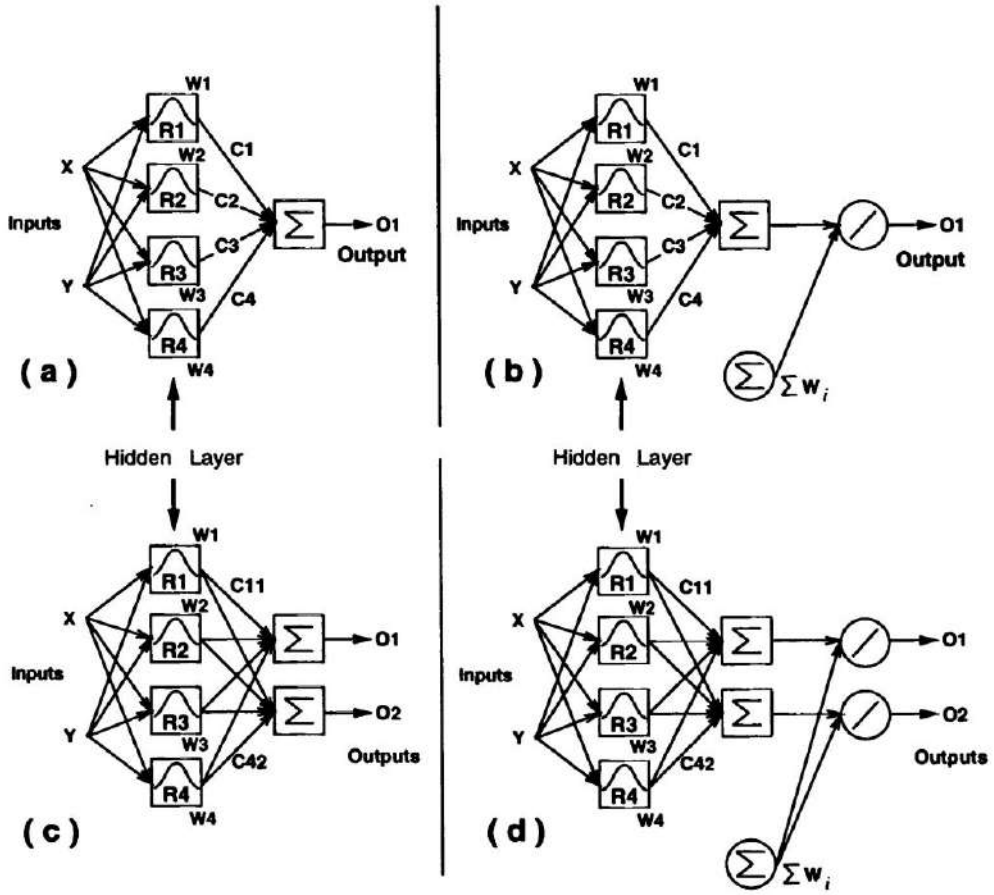


Figure 9.10. *Four RBFNs that possess four basis functions: (a) single-output RBFN that uses weighted sum; (b) single-output RBFN that uses weighted average; (c) two-output RBFN that uses weighted sum; (d) two-output RBFN that uses weighted average. The network in (d) is equivalent to Figure 13.3 (upper right). [Note that in (b) and (d), four connections to the lower summation unit are omitted for simplicity.]*

that this extended version performed better than the original RBFN with the same number of fitting parameters.

An RBFN's approximation capacity may be further improved with supervised adjustments of the center and shape of the receptive field (or radial basis) functions [24, 60]. Several learning algorithms have been proposed to identify the parameters ($\mathbf{u}_i$, σ_i, and c_i) of an RBFN. Besides using a supervised learning scheme alone to update all modifiable parameters, a variety of sequential training algo-

layer. Typically, $R_i(\cdot)$ is a Gaussian function

$$R_i(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{u}_i\|^2}{2\sigma_i^2}\right) \tag{9.14}$$

or a logistic function

$$R_i(\mathbf{x}) = \frac{1}{1 + \exp[\|\mathbf{x} - \mathbf{u}_i\|^2/\sigma_i^2]}. \tag{9.15}$$

Thus, the activation level of radial basis function w_i computed by the ith hidden unit is maximum when the input vector $\mathbf{x}$ is at the center $\mathbf{u}_i$ of that unit.

The output of an RBFN can be computed in two ways. In the simpler method, as shown in Figure 9.10(a), the final output is the **weighted sum** of the output value associated with each receptive field:

$$d(\mathbf{x}) = \sum_{i=1}^{H} c_i w_i = \sum_{i=1}^{H} c_i R_i(\mathbf{x}), \tag{9.16}$$

where c_i is the output value associated with the ith receptive field. We can also view c_i as the connection weight between the receptive field i and the output unit. A more complicated method for calculating the overall output is to take the **weighted average** of the output associated with each receptive field:

$$d(\mathbf{x}) = \frac{\sum_{i=1}^{H} c_i w_i}{\sum_{i=1}^{H} w_i} = \frac{\sum_{i=1}^{H} c_i R_i(\mathbf{x})}{\sum_{i=1}^{H} R_i(\mathbf{x})}. \tag{9.17}$$

Weighted average has a higher degree of computational complexity, but it is advantageous in that points in the areas of overlap between two or more receptive fields will have a well-interpolated overall output between the outputs of the overlapping receptive fields. An example is presented in Section 9.5.4.

For representation purposes, if we change the radial basis function $R_i(\mathbf{x})$ in each node of layer 2 in Figure 9.10(a) to its **normalized** counterpart $R_i(\mathbf{x})/\sum_i R_i(\mathbf{x})$, then the overall output is specified by Equation (9.17). A more explicit representation is shown in Figure 9.10(b), where the division of the weighted sum $(\sum_i c_i w_i)$ by the activation total $(\sum_i w_i)$ is indicated in the division node in the last layer. In Figure 9.10, plots (c) and (d) are the two-output counterparts of the RBFNs in (a) and (b).

Moody-Darken's RBFN may be extended by assigning a linear function to the output function of each receptive field—that is, making c_i a linear combination of the input variables plus a constant:

$$c_i = \mathbf{a}_i^T \mathbf{x} + b_i, \tag{9.18}$$

where $\mathbf{a}_i$ is a parameter vector and b_i is a scalar parameter. Stokbro et al. [57] used this structure to model the Mackey-Glass chaotic time series [27] and found

layer. Typically, $R_i(\cdot)$ is a Gaussian function

$$R_i(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{u}_i\|^2}{2\sigma_i^2}\right) \tag{9.14}$$

or a logistic function

$$R_i(\mathbf{x}) = \frac{1}{1 + \exp[\|\mathbf{x} - \mathbf{u}_i\|^2/\sigma_i^2]}. \tag{9.15}$$

Thus, the activation level of radial basis function w_i computed by the ith hidden unit is maximum when the input vector $\mathbf{x}$ is at the center $\mathbf{u}_i$ of that unit.

The output of an RBFN can be computed in two ways. In the simpler method, as shown in Figure 9.10(a), the final output is the **weighted sum** of the output value associated with each receptive field:

$$d(\mathbf{x}) = \sum_{i=1}^{H} c_i w_i = \sum_{i=1}^{H} c_i R_i(\mathbf{x}), \tag{9.16}$$

where c_i is the output value associated with the ith receptive field. We can also view c_i as the connection weight between the receptive field i and the output unit. A more complicated method for calculating the overall output is to take the **weighted average** of the output associated with each receptive field:

$$d(\mathbf{x}) = \frac{\sum_{i=1}^{H} c_i w_i}{\sum_{i=1}^{H} w_i} = \frac{\sum_{i=1}^{H} c_i R_i(\mathbf{x})}{\sum_{i=1}^{H} R_i(\mathbf{x})}. \tag{9.17}$$

Weighted average has a higher degree of computational complexity, but it is advantageous in that points in the areas of overlap between two or more receptive fields will have a well-interpolated overall output between the outputs of the overlapping receptive fields. An example is presented in Section 9.5.4.

For representation purposes, if we change the radial basis function $R_i(\mathbf{x})$ in each node of layer 2 in Figure 9.10(a) to its **normalized** counterpart $R_i(\mathbf{x})/\sum_i R_i(\mathbf{x})$, then the overall output is specified by Equation (9.17). A more explicit representation is shown in Figure 9.10(b), where the division of the weighted sum $(\sum_i c_i w_i)$ by the activation total $(\sum_i w_i)$ is indicated in the division node in the last layer. In Figure 9.10, plots (c) and (d) are the two-output counterparts of the RBFNs in (a) and (b).

Moody-Darken's RBFN may be extended by assigning a linear function to the output function of each receptive field—that is, making c_i a linear combination of the input variables plus a constant:

$$c_i = \mathbf{a}_i^T \mathbf{x} + b_i, \tag{9.18}$$

where $\mathbf{a}_i$ is a parameter vector and b_i is a scalar parameter. Stokbro et al. [57] used this structure to model the Mackey-Glass chaotic time series [27] and found

0.25 (see dashed lines in Figure 13.8), so the error signals at front-end layers tend to be smaller due to multiple multiplication of this term. Therefore, the learning rate η of the front-end layers should be larger than that of the output layer. This is called the **learning rate rescaling**; see ref. [46] for details.

9.4.3 MLP's Approximation Power

The approximation power of backpropagation MLPs has been explored by some researchers. Yet there is very little theoretical guidance for determining network size in terms of, say, the number of hidden nodes and hidden layers it should contain. Cybenko [8] showed that a backpropagation MLP, with one hidden layers and any fixed continuous sigmoidal nonlinear function, can approximate any continuous function arbitrarily well on a compact set. When used as a binary-valued neural network with the hard-limiter (step) activation function, a backpropagation MLP with two hidden layers can form arbitrary complex decision regions to separate different classes, as Lippmann [25] pointed out. For function approximation as well as data classification, two hidden layers may be required to learn a piecewise-continuous function [29]. In their book, Hertz et al. [13] introduced an intuitive explanation that MLPs with two hidden layers may be able to construct localized receptive fields out of logistic functions. Thus, two-layer MLPs may have abilities comparable to radial basis function networks, which are discussed next.

9.5 RADIAL BASIS FUNCTION NETWORKS

9.5.1 Architectures and Learning Methods

Locally tuned and overlapping receptive fields are well-known structures that have been studied in regions of the cerebral cortex, the visual cortex, and others. Drawing on knowledge of biological receptive fields, Moody and Darken [34, 35] proposed a network structure that employs local receptive fields to perform function mappings. Similar schemes have been proposed by Powell [44], Broomhead and Lowe [4], and many others in the areas of **interpolation** and **approximation theory**; these schemes are collectively called radial basis function approximations. Here we shall call the network structure the **radial basis function network** or **RBFN**.

Figure 9.10(a) shows a schematic diagram of an RBFN with four receptive field units; the activation level of the ith receptive field unit (or hidden unit) is

$$w_i = R_i(\mathbf{x}) = R_i(\|\mathbf{x} - \mathbf{u}_i\|/\sigma_i), \; i = 1, \; 2, \; \ldots, \; H, \tag{9.13}$$

where $\mathbf{x}$ is a multidimensional input vector, $\mathbf{u}_i$ is a vector with the same dimension as $\mathbf{x}$, H is the number of radial basis functions (or, equivalently, receptive field units), and $R_i(\cdot)$ is the ith radial basis function with a single maximum at the origin. There are no connection weights between the input layer and the hidden

due to gradient noise or high spatial frequencies in the error surface. However, the use of momentum terms does not always seem to speed up training; it is more or less application dependent [63].

Another useful technique is normalized weight updating:

$$\Delta \mathbf{w} = -\kappa \frac{\nabla_{\mathbf{w}} E}{\|\nabla_{\mathbf{w}} E\|}. \tag{9.12}$$

This causes the network's weight vector to move the same Euclidean distance κ in the weight space with each update, which allows control of the distance κ based on the history of error measures. One of the adaptation strategies for varying the step size κ is explained in Section 6.7.2 of Chapter 6. Other methods for speeding up MLP backpropagation training include the quick-propagation algorithm [9], the delta-bar-delta approach [14], the extended Kalman filter method [54, 55], second-order optimization [58], and the optimal filtering approach [53].

Generally, an MLP with hyperbolic tangent functions can be trained more rapidly than one with logistic functions. This is only an empirical observation and it has exceptions (for instance, the encoding problem in ref. [9]), but it is advisable to try both types of MLP when encountering a new application.

Sometimes it is desirable to do **data scaling** on the raw training data and then use the processed data to train the MLP. In **output scaling**, the range of target values is constrained to remain within the range of the sigmoidal activation function. For instance, for an MLP with hyperbolic tangent functions, the target values must be within, say, $[-0.9, 0.9]$, instead of within the usual activation function range $[-1, 1]$. This prevents backpropagation from driving some of the connection weights to infinity and slowing down training. A similar approach using *modified sigmoid functions* is discussed in Section 13.3.3. In **input scaling**, the range of each input is scaled (usually linearly) to the range of the activation function used. This scaling allows the connection weights to have the same order of magnitude during training.

The initial values of the connection weights and biases in an MLP should be uniformly distributed across a small range, usually $[-1, 1]$. If the initial values of these modifiable parameters are too large, then some of the neurons might get saturated and produce small error signals. On the other hand, if the range is too small, then the gradient vector is also small and learning will be very slow initially. Note that when all the free parameters are zeros, the gradient vector is always zero since it happens to be a *saddle point* in the error landscape. Another and better way to initialize network parameters is to choose the weight and bias values such that the "slope" parts of neurons in the hidden layer can cover the input space; see ref. [37] for details.

All neurons in an MLP should get updated at approximately the same rate. However, the error signals at the the output layer tend to be larger than those at the front-end layer of the network. This can be seen directly in Equation (9.7), where $x_i(1 - x_i)$ appears once in the output layer, twice in the layer next to the output, and so on. Note that the term $x_i(1 - x_i)$ is always less than or equal to

where d_k is the desired output for node k, and x_k is the actual output for node k when the input part of the pth data pair is presented. To find the gradient vector, an error term $\bar{\epsilon}_i$ for node i is defined as

$$\bar{\epsilon}_i = \frac{\partial^+ E_p}{\partial \bar{x}_i}. \tag{9.6}$$

By the chain rule, the recursive formula for $\bar{\epsilon}_i$ can be written as

$$\bar{\epsilon}_i = \begin{cases} -2(d_i - x_i)\dfrac{\partial x_i}{\partial \bar{x}_i} = -2(d_i - x_i)x_i(1 - x_i) & \text{if node } i \text{ is a output node,} \\ \dfrac{\partial x_i}{\partial \bar{x}_i} = \sum_{j,i<j} \dfrac{\partial^+ E_p}{\partial \bar{x}_j}\dfrac{\partial \bar{x}_j}{\partial x_i} = x_i(1 - x_i)\sum_{j,i<j} \bar{\epsilon}_j w_{ij} & \text{otherwise,} \end{cases} \tag{9.7}$$

where w_{ij} is the connection weight from node i to j; and w_{ij} is zero if there is no direct connection. Then the weight update w_{ki} for on-line (pattern-by-pattern) learning is

$$\Delta w_{ki} = -\eta \frac{\partial^+ E_p}{\partial w_{ki}} = -\eta \frac{\partial^+ E_p}{\partial \bar{x}_i}\frac{\partial \bar{x}_i}{\partial w_{ki}} = -\eta \bar{\epsilon}_i x_k, \tag{9.8}$$

where η is a learning rate that affects the convergence speed and stability of the weights during learning. The update formula for the bias of each node can be derived similarly.

For off-line (batch) learning, the connection weight w_{ki} is updated only after presentation of the entire data set, or only after an **epoch**:

$$\Delta w_{ki} = -\eta \frac{\partial^+ E}{\partial w_{ki}} = -\eta \sum_p \frac{\partial^+ E_p}{\partial w_{ki}}, \tag{9.9}$$

or, in vector form,

$$\Delta \mathbf{w} = -\eta \frac{\partial^+ E}{\partial \mathbf{w}} = -\eta \nabla_{\mathbf{w}} E, \tag{9.10}$$

where $E = \sum_p E_p$. This corresponds to a way of using the true gradient direction based on the entire data set.

9.4.2 Methods of Speeding Up MLP Training

Quite a few ad hoc methods exist to speed up MLP backpropagation training. Some of them are applicable to general backpropagation gradient descent, while others are only effective for MLP backpropagation.

One way to speed up off-line training is to use the so-called **momentum term** [48]:

$$\Delta \mathbf{w} = -\eta \nabla_{\mathbf{w}} E + \alpha \Delta \mathbf{w}_{\text{prev}}, \tag{9.11}$$

where $\mathbf{w}_{\text{prev}}$ is the previous update amount, and the **momentum constant** α, in practice, is usually set to something between 0.1 and 1. The addition of the momentum term smoothes weight updating and tends to resist erratic weight changes

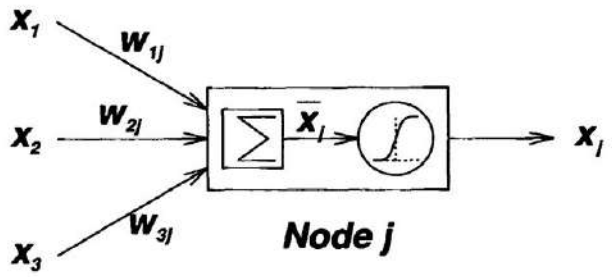


Figure 9.8. *Node j of a backpropagation MLP.*

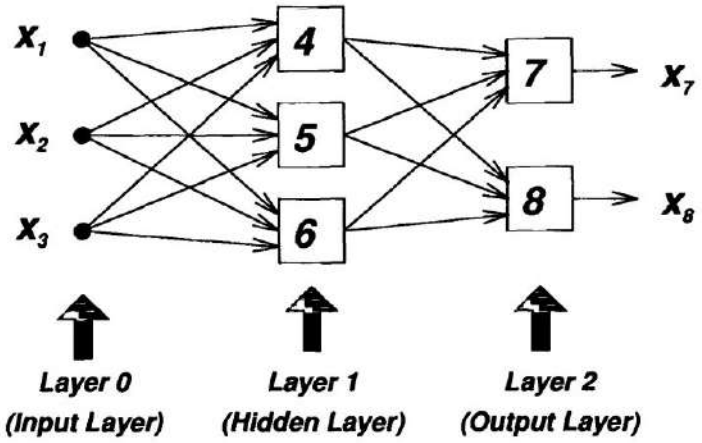


Figure 9.9. *A 3-3-2 backpropagation MLP.*

the weight associated with the link connecting nodes i and j, and w_j is the bias of node j. Since the weights w_{ij} are actually internal parameters associated with each node j, changing the weights of a node will alter the behavior of the node and in turn alter the behavior of the whole backpropagation MLP. Figure 9.9 shows a two-layer backpropagation MLP with three inputs to the input layer, three neurons in the hidden layer, and two output neurons in the output layer. For simplicity, this backpropagation MLP will be referred to as a 3-3-2 network, corresponding to the number of nodes in each layer. (Note that the input layer is composed of three buffer nodes for distributing the input signals; therefore, this layer is conventionally not counted as a physical layer of a backpropagation MLP.)

The **backward error propagation**, also known as the **backpropagation** (**BP**) or the **generalized delta rule** (**GDR**), is explained next. First, a squared error measure for the pth input-output pair is defined as

$$E_p = \sum_k (d_k - x_k)^2, \tag{9.5}$$

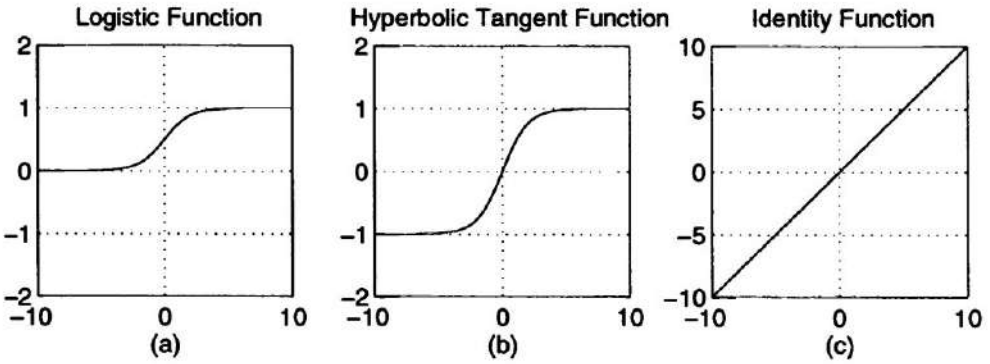


Figure 9.7. *Activation functions for backpropagation MLPs: (a) logistic function; (b) hyperbolic tangent function; (c) identity function.* (MATLAB file: `activati.m`)

referred to as *binary* sigmoidal.) Both of these activations are used often on regression and classification problems. Other activation functions are discussed in Section 13.3.3 of Chapter 13.

For a neural network to approximate a continuous-valued function not limited to the interval $[0, 1]$ or $[-1, 1]$, we usually let the node function for the output layer be a weighted sum with no squashing functions. This is equivalent to a situation in which the activation function is an identity function, and output nodes of this type are often called **linear nodes.**

Backpropagation MLPs are by far the most commonly used NN structures for applications in a wide range of areas, such as pattern recognition, signal processing, data compression, and automatic control. Some of the well-known instances of applications include NETtalk [51, 52], which trained an MLP to pronounce English text, Carnegie Mellon University's ALVINN (Autonomous Land Vehicle in a Neural Network) [42, 43], which used an MLP for steering an autonomous vehicle; and optical character recognition (OCR) [23, 49].

9.4.1 Backpropagation Learning Rule

For simplicity, we assume that the backpropagation MLP in question uses the logistic function as its activation function; the reader is encouraged to derive the backpropagation procedure when other types of continuous activation functions are used.

The **net input** $\bar{x}$ of a node is defined as the weighted sum of the incoming signals plus a bias term. For instance, the net input and output of node j in Figure 9.8 are

$$\begin{aligned} \bar{x}_j &= \textstyle\sum_i w_{ij} x_i + w_j, \\ x_j &= f(\bar{x}_j) = \frac{1}{1 + \exp(-\bar{x}_j)}, \end{aligned} \tag{9.4}$$

where x_i is the output of node i located in any one of the previous layers, w_{ij} is

were connected to an AND logic device (Madaline unit) to provide an output [65]. However, the Adaline and Madaline systems were limited in that they had only one layer with adjustable weights, just like single-layer perceptrons.

Adaline and Madline have been used for adaptive noise cancellation [61] and adaptive inverse control [64]. In adaptive noise cancellation, the objective is to filter out an interference component by identifying a linear model of a measurable noise source and the corresponding unmeasurable interference; such applications include interference canceling in electrocardiograms (ECGs), echo elimination from long-distance telephone transmission lines, and antenna sidelobe interference canceling [64]; for more details on adaptive inverse control, see refs. [64] or Section 17.4 of this text. For details on Adaline, Madline, and LMS methods, refer to refs. [64] and [63].

9.4 BACKPROPAGATION MULTILAYER PERCEPTRONS

As mentioned earlier, the lack of suitable training methods for **multilayer perceptrons (MLPs)** led to a waning of interest in neural networks in the 1960s and 1970s. This was not changed until the reformulation of the **backpropagation** training method for MLPs in the mid-1980s by Rumelhart et al. [48]. (The derivation of the backpropagation method in a more general framework can be found in Section 8.3 of Chapter 8.)

The single-layer perceptron discussed in Section 9.2 is a principal NN component and provides the grounds for current understanding and most applications of NNs. However, because of the nondifferentiability of the hard-limiter activation function, the learning strategies of early multilayer perceptrons with signum or step activation functions are not obvious unless continuous activation functions are employed.

A backpropagation MLP, as already mentioned in Examples 8.3 and 8.4, is an adaptive network whose nodes (or neurons) perform the same function on incoming signals; this node function is usually a composite of the weighted sum and a differentiable nonlinear activation function, also known as the **transfer function**. Figure 9.7 depicts three of the most commonly used activation functions in backpropagation MLPs:

$$\begin{aligned}
&\text{Logistic function:} && f(x) = \frac{1}{1+e^{-x}} \\
&\text{Hyperbolic tangent function:} && f(x) = \tanh(x/2) = \frac{1-e^{-x}}{1+e^{-x}} \\
&\text{Identity function:} && f(x) = x.
\end{aligned}$$

Both the hyperbolic tangent and logistic functions approximate the signum and step function, respectively, and yet provide smooth, nonzero derivatives with respect to input signals. Sometimes these two activation functions are referred to as **squashing functions** since the inputs to these functions are squashed to the range $[0, 1]$ or $[-1, 1]$. They are also called **sigmoidal functions** because their S-shaped curves exhibit smoothness and asymptotic properties. (Sometimes the hyperbolic tangent function are referred to as ***bipolar*** sigmoidal and the logistic function are

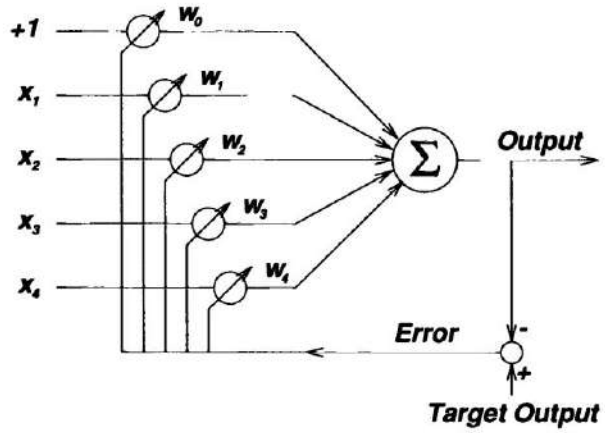


Figure 9.6. *Adaline (adaptive linear element).*

where t_p is the target output and o_p is the actual output of the Adaline. The derivative of E_p with respect to each weight w_i is

$$\frac{\partial E_p}{\partial w_i} = -2(t_p - o_p)x_i.$$

Therefore, to decrease E_p by gradient descent, the update formula for w_i on the pth input-output pattern is

$$\Delta_p w_i = \eta(t_p - o_p)x_i. \tag{9.3}$$

This update formula has strong intuitive appeal. It essentially states that when $t_p > o_p$, we want to boost o_p by increasing $w_i x_i$; therefore, we should increase w_i if x_i is positive and decrease w_i if x_i is negative. Similar reasoning holds when $t_p < o_p$. Since the delta rule tries to minimize squared errors, it is also referred to as the **least mean square (LMS) learning procedure** or **Widrow-Hoff learning rule**. The features of the delta rule are as follows:

- Simplicity: This is obvious from Equation (9.3).
- Distributed learning: Learning is not reliant on central control of the network; it can be performed locally at each node level; see Equation (9.3).
- On-line (or pattern-by-pattern) learning: Weights are updated after presentation of each pattern.

These features make Adaline, with the delta rule, suitable for simple hardware implementation.

In the 1960s, two or more Adaline components were integrated to develop **Madaline** (many Adalines) in an attempt to implement nonlinearly separable logic functions. To solve the previously mentioned XOR problem, for instance, two Adalines

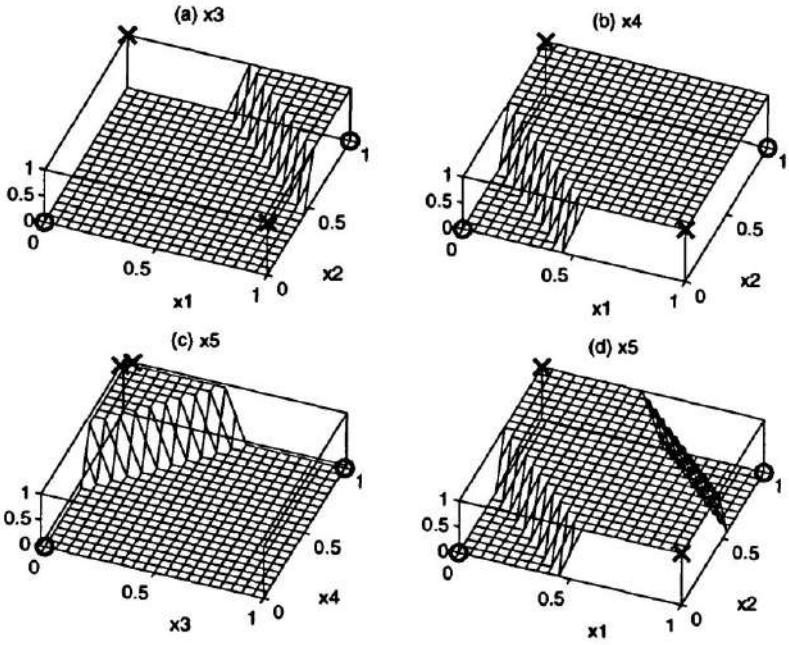


Figure 9.5. *Node outputs as surfaces of their inputs in Figure 9.4: (a) x_3; (b) x_4; (c) x_5 (as a function of x_3 and x_4); (d) x_5 (as a function of x_1 and x_2).* (MATLAB file: `xorsurf1.m`)

is a weighted linear combination of the inputs plus a constant term:

$$o = \sum_{i=1}^{n} w_i x_i + w_0. \tag{9.2}$$

In a simple physical implementation, the input signals x_i are voltages and the w_i are conductances of controllable resistors; the network's output is the summation of the currents caused by the input voltages. The problem is to find a suitable set of conductances (or weights) such that the input-output behavior of the Adaline is close to a set of desired input-output data points.

It is obvious that the preceding Adaline equation is an exactly linear model with $n + 1$ linear parameters, so we can employ the least-squares methods introduced in Chapter 5 to minimize the error in the sense of least squares. However, most of the least-squares methods require extensive calculations, which are not possible in a physical system with simple components. To overcome this, Widrow and Hoff introduced the **delta rule** for adjusting the weights. For the pth input-output pattern, the error measure of a single-output Adaline can be expressed as

$$E_p = (t_p - o_p)^2,$$

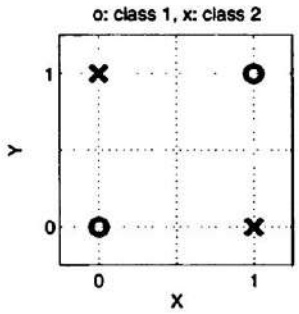


Figure 9.3. *XOR problem.* (MATLAB file: `xordata.m`)

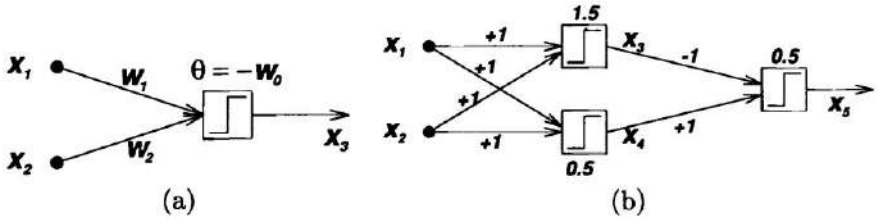


Figure 9.4. *Perceptrons for the two-input exclusive-OR problem: (a) the single-layer perceptron, and (b) the two-layer perceptron. Both use the step function as the activation function for each node.*

However, the set of inequalities is self-contradictory when considered as a whole.

It is possible to solve the problem with the two-layer perceptron illustrated in Figure 9.4(b), in which the connection weights and thresholds are indicated. More specifically, we can plot the output of each neuron as a surface of its two inputs, as shown in Figures 9.5(a) through 9.5(d). Figure 9.5(d) is the overall input-output plot of the two-layer perceptron, indicating that it has solved the XOR problem.

In summary, multilayer perceptrons can solve nonlinearly separable problems and are thus much more powerful than the original single-layer version. In Section 9.4, we discuss a learning method that can be used to find appropriate connection weights and thresholds in multilayer perceptrons.

9.3 ADALINE

The **adaptive linear element** (or **Adaline**), suggested by Widrow and Hoff [62], represents a classical example of the simplest intelligent self-learning system that can adapt itself to achieve a given modeling task. Figure 9.6 is a schematic diagram for such a network. It has a purely linear output unit; hence the network output o

The foregoing learning rule can be applied as well to updating a threshold θ ($= -w_0$) according to Equation (9.1). The value for the learning rate η can be a constant throughout training; or it can be a varying quantity proportional to the error. An η that is proportional to the error usually leads to faster convergence but can cause unstable learning.

The preceding learning algorithm above is roughly based on gradient descent; Rosenblatt [47] proved that there exists a method for tuning the weights that is guaranteed to converge to provide the required output if and only if such a set of weights exist. This is called the **perceptron convergence theorem**. Moreover, depending on the functions g_i, perceptrons can be grouped into different families; a number of these families and their properties are described in refs. [47].

In the early 1960s, perceptrons created a great deal of interest and optimism directed toward building real self-learning intelligent systems. However, the initial enthusiasm waned after the publication of Minsky and Papert's *Perceptrons* [33] in 1969, in which they analyzed the perceptron extensively and concluded that single-layer perceptrons can only be used for toy problems. One of their most discouraging results shows that a single-layer perceptron cannot represent a simple exclusive-OR function, as explained next.

9.2.2 Exclusive-OR Problem

The simplest and most well-known pattern recognition problem in neural network literature is the **exclusive-OR (XOR)** problem. The task is to classify a binary input vector to class 0 if the vector has an even number of 1's, or assign it to class 1. For a two-input binary XOR problem, the desired behavior is regulated by a truth table:

	X	Y	Class
Desired i/o pair 1	0	0	0
Desired i/o pair 2	0	1	1
Desired i/o pair 3	1	0	1
Desired i/o pair 4	1	1	0

A bipolar XOR problem is similarly defined except that all instances of 1 in the truth table are replaced with -1.

The XOR problem is not **linearly separable**; this can easily be observed from the plot in Figure 9.3. In other words, we cannot use a single-layer perceptron [Figure 9.4(a)] to construct a straight line to partition the two-dimensional input space into two regions, each containing only data points of the same class. Symbolically, using a single-layer perceptron to solve this problem requires satisfying the following four inequalities:

$$\begin{aligned}
0 \times w_1 + 0 \times w_2 + w_0 \leq 0 &\iff w_0 \leq 0, \\
0 \times w_1 + 1 \times w_2 + w_0 > 0 &\iff w_0 > -w_2, \\
1 \times w_1 + 0 \times w_2 + w_0 > 0 &\iff w_0 > -w_1, \\
1 \times w_1 + 1 \times w_2 + w_0 \leq 0 &\iff w_0 \leq -w_1 - w_2.
\end{aligned}$$

making adjustments to the relevant connection strengths (i.e., weights w_i) and a threshold value θ. For a two-class problem (for instance, determining whether the given pattern in Figure 9.1 is a "P" or not), the output layer usually has only a single node. For an n-class problem with n greater than or equal to 3, the output layer usually has n nodes, each corresponding to a class, and the output node with the largest value indicates which class the input vector belongs to.

Each function g_i in layer 1 is a fixed function that has to be determined *a priori*; it maps all or a part of the input pattern into a **binary** value $x_i \in \{-1, 1\}$ or a **bipolar** value $x_i \in \{0, 1\}$. The term x_i is referred to as **active** or **excitatory** if its value is 1, **inactive** if its value is 0, and **inhibitory** if its value is -1. The output unit is a **linear threshold element** with a threshold value θ:

$$\begin{aligned} o &= f\left(\sum_{i=1}^{n} w_i x_i - \theta\right), \\ &= f\left(\sum_{i=1}^{n} w_i x_i + w_0\right), \; w_0 \equiv -\theta, \\ &= f\left(\sum_{i=0}^{n} w_i x_i\right), \; x_0 \equiv 1. \end{aligned} \tag{9.1}$$

where w_i is a modifiable weight associated with an incoming signal x_i; and w_0 ($= -\theta$) is the bias term. For computational efficiency, we can introduce the bias connection weight w_0 in place of the threshold value θ; Equation (9.1) shows that the threshold can be viewed as the connection weight between the output unit and a "dummy" incoming signal x_0 that is always equal to 1, as illustrated in Figure 9.2. In Equation (9.1), $f(\cdot)$ is the **activation function** of the perceptron and it is typically either a **signum function** $\mathrm{sgn}(x)$ or **step function** $\mathrm{step}(x)$:

$$\mathrm{sgn}(x) = \begin{cases} 1 & \text{if } x > 0, \\ -1 & \text{otherwise,} \end{cases}$$

$$\mathrm{step}(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

Note that the "feature detector" g_i can be any function of the input pattern, but the learning procedure only adjusts the connection weights to the output unit (in the last layer). Since only the weights leading to the last layer are modifiable, the perceptron in Figure 9.1 is usually treated as a **single-layer perceptron**. Starting with a set of random connection weights, the basic learning algorithm for a single-layer perceptron repeats the following steps until the weights converge:

1. Select an input vector $\mathbf{x}$ from the training data set.

2. If the perceptron gives an incorrect response, modify all connection weights w_i according to

$$\Delta w_i = \eta t_i x_i,$$

where t_i is a target output and η is a learning rate.

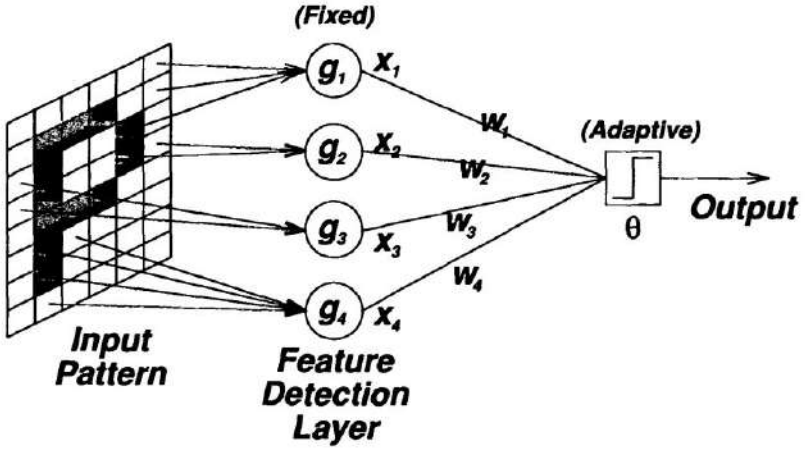


Figure 9.1. *The perceptron.*

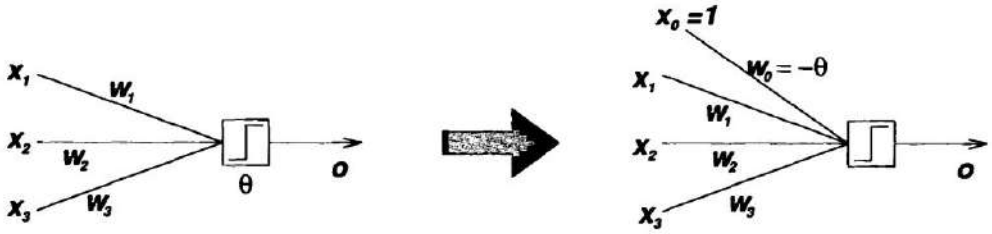


Figure 9.2. *Introduction of the bias connection weight; the bias term* w_0 $(= -\theta)$ *can be viewed as the connection weight between the output unit and a "dummy" incoming signal* x_0 *that is always equal to 1.*

9.2 PERCEPTRONS

9.2.1 Architecture and Learning Rule

The **perceptron** represents one of the early attempts to build intelligent and self-learning systems using simple components. It was derived from a biological brain neuron model introduced by McCulloch and Pitts [32] in 1943. Later, Rosenblatt [47] designed the perceptron with a view toward explaining and modeling pattern-recognition abilities of biological visual systems. Although the goal is ambitious, the network paradigm is simple. Figure 9.1 is a typical perceptron setup for pattern-recognition applications, in which visual patterns are represented as matrices of elements between 0 and 1. The first layer of the perceptron acts as a set of "feature detectors" that are hardwired to the input signals to detect specific features. The second (output) layer takes the outputs of the "feature detectors" in the first layer and classifies the given input pattern. Learning is initiated by

Chapter 9

Supervised Learning Neural Networks

J.-S. R. Jang and E. Mizutani

9.1 INTRODUCTION

Artificial neural networks, or simply **neural networks** (**NNs**), have been studied for more than three decades since Rosenblatt [47] first applied single-layer *perceptrons* to pattern classification learning in the late 1950s. However, because Minsky and Papert [33] pointed out that single-layer systems were limited and expressed pessimism over multilayer systems, interest in NNs dwindled in the 1970s. The recent resurgence of interest in the field of NNs has been inspired by new developments in NN learning algorithms [10, 40, 48, 59], analog VLSI (very large scale integrated) circuits, and parallel processing techniques [25].

Quite a few NN models have been proposed and investigated in recent years. These NN models can be classified according to various criteria, such as their learning methods (supervised versus unsupervised), architectures (feedforward versus recurrent), output types (binary versus continuous), node types (uniform versus hybrid), implementations (software versus hardware), connection weights (adjustable versus hardwired), operations (biologically motivated versus psychologically motivated), and so on. In this chapter, we confine our scope to modeling problems with desired input-output data sets, so the resulting networks must have adjustable parameters that are updated by a supervised learning rule. Such networks are often referred to as **supervised learning** or **mapping networks**, since we are interested in shaping the input-output mappings of the networks according to a given training data set. (For details on unsupervised learning networks that try to cluster a given data set, see the next chapter.)

Figure 11.9. *One-layer single-output network with Hebbian learning for principal component analysis.*

A sequence of learning patterns indexed by p is assumed here to be presented to the network; in addition, all initial weights are *zero*. By using Equation (11.6), the update amount of a weight after the entire data set is presented will be

$$w_{ij} = \eta \sum_p y_{ip} y_{jp}. \tag{11.9}$$

Frequent input patterns have the most impact on the weights and, eventually, cause the network to produce the largest outputs. Applying the plain Hebbian learning in Equation (11.6) causes unconstrained growth of the weights. Hence, in some cases, the Hebbian rule is modified to counteract the unlimited growth of weights. Weight normalization, as described in subsection 11.6.2, is one such method.

The rationale behind the Hebbian learning rule is most easily understood via a single-layer n-input one-output neural network with identity activation functions, as shown in Figure 11.9. The output y is equal to $\sum_{i=1}^{n} w_i x_i$, or in matrix form,

$$y = \mathbf{w}^T \mathbf{x} = \mathbf{x}^T \mathbf{w},$$

where $\mathbf{x} = [x_1, \ldots, x_n]^T$ is the input vector and $\mathbf{w} = [w_1, \ldots, w_n]^T$ is the weight vector. The corresponding Hebbian learning rule is

$$\Delta \mathbf{w} = \eta y \mathbf{x}. \tag{11.10}$$

The preceding learning rule is an on-line steepest-descent scheme that minimizes the objective function

$$J = \frac{1}{2} \sum_p y_p^2 = \frac{1}{2} \sum_p (\mathbf{x}_p^T \mathbf{w})^2, \tag{11.11}$$

where $\mathbf{x}_p$ and y_p are the pth input and corresponding desired output, respectively. If $\mathbf{w}$ is a unit vector, then $\mathbf{x}_p^T \mathbf{w}$ is the projection of the vector $\mathbf{x}_p$ onto the direction

specified by a vector $\mathbf{w}$. The minimization of J implies finding a unit-length $\mathbf{w}$ that most accurately represents the *direction* of the entire data set.

Other learning algorithms (such as in the Hopfield network learning and supervised correlation learning) often reflect the Hebbian learning principle [39]; they share the correlational property in their formulas.

11.5 HEBBIAN LEARNING

Hebb [20] described a simple learning method of synaptic weight change. When two cells fire simultaneously (i.e., have strong responses), their connection strength (or weight) increases. Such phenomenon is the so-called **Hebbian learning**, where the weight increase between two neurons is proportional to the frequency at which they fire together. Among various mathematical formulas of this principle, the simplest one is expressed as

$$\Delta w_{ij} = \eta y_i y_j, \tag{11.6}$$

where η is the learning rate. Since weights are adjusted according to the correlation of neuron outputs, the preceding formula is a type of *correlational learning rule*. The ith neuron's output y_i can be regarded as an input (x_i) to another neuron j (Figure 11.8), so Equation (11.6) can be written as

$$\Delta w_{ij} = \eta y_j x_i. \tag{11.7}$$

Restated, a weight is assumed to change proportionately to the *correlation* of the input and output signals. By using a neuron function $f(\cdot)$, y_j is given by

$$y_j = f(\mathbf{w}_j^T \mathbf{x}).$$

Thus, Equation (11.7) is equivalent to the following:

$$\Delta w_{ij} = \eta\, f(\mathbf{w}_j^T \mathbf{x})\, x_i. \tag{11.8}$$

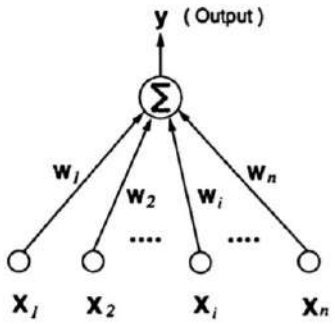


Figure 11.9. *One-layer single-output network with Hebbian learning for principal component analysis.*

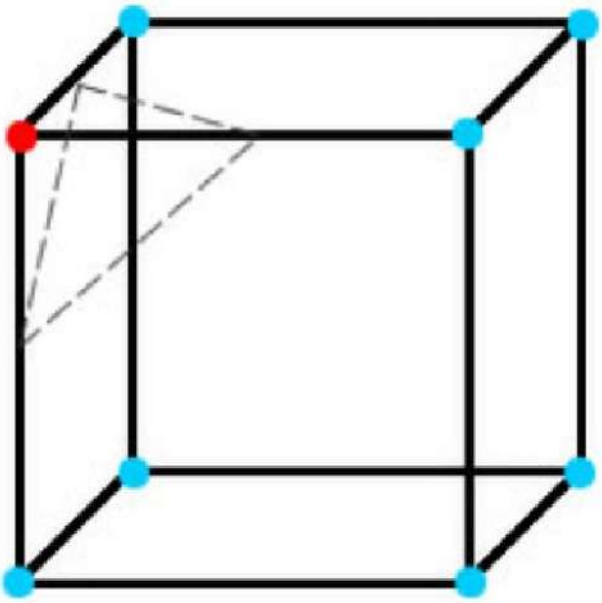
A sequence of learning patterns indexed by p is assumed here to be presented to the network; in addition, all initial weights are *zero*. By using Equation (11.6), the update amount of a weight after the entire data set is presented will be

$$w_{ij} = \eta \sum_p y_{ip} y_{jp}. \tag{11.9}$$

Frequent input patterns have the most impact on the weights and, eventually, cause the network to produce the largest outputs. Applying the plain Hebbian learning in Equation (11.6) causes unconstrained growth of the weights. Hence, in some cases, the Hebbian rule is modified to counteract the unlimited growth of weights. Weight

Three dimensions

Extending the above example to three dimensions. You need a plane for separating the two classes.

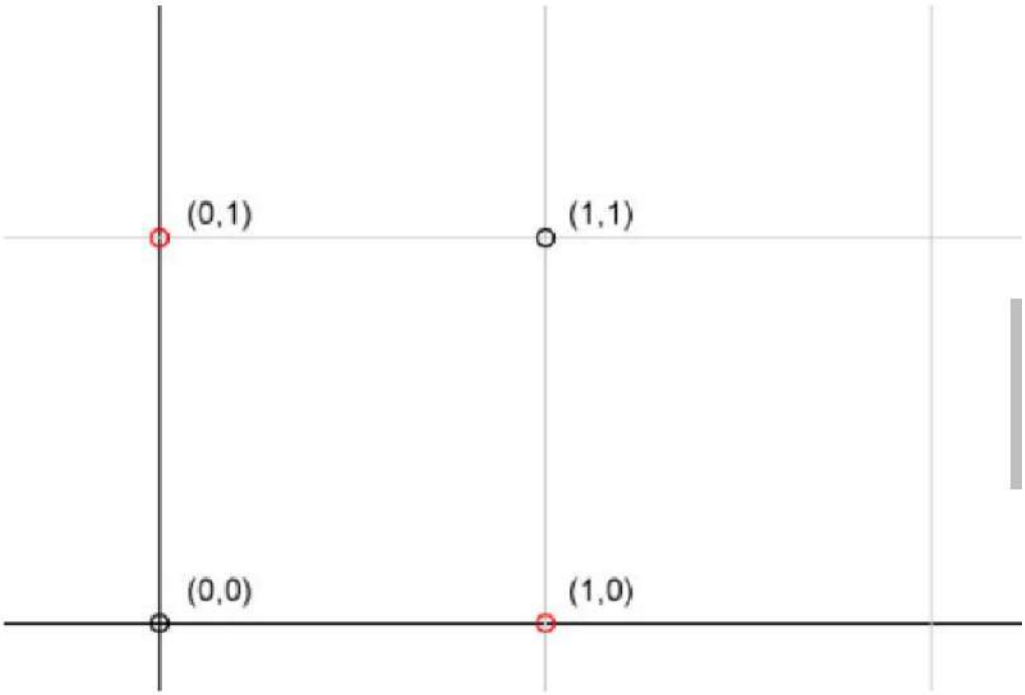


Linear separability in 3D space

The dashed plane separates the red point from the other blue points. So its linearly separable. If bottom right point on the opposite side was red too, it would become linearly inseparable .

Extending to n dimensions

Now consider this:



Linearly inseparable

In this case, you just cannot use one single line to separate the two classes (one containing the black points and one containing the red points). So, they are linearly inseparable.

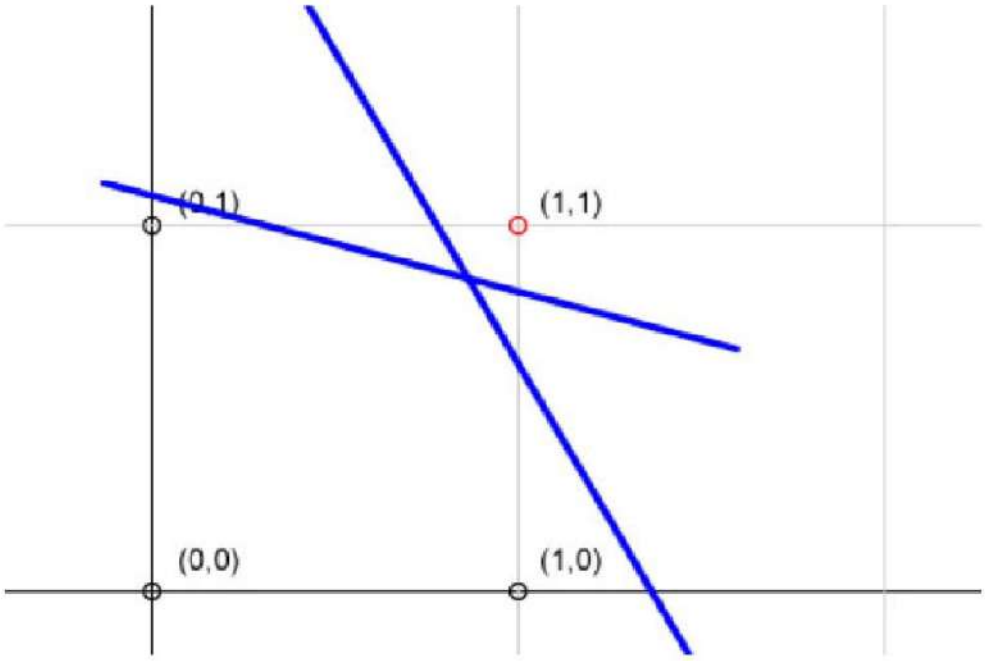
Three dimensions

Extending the above example to three dimensions. You need a plane for separating the two classes.

separate something from itself)

Two Dimensions

On extending this idea to two dimensions, some more possibilities come into existence. Consider the following:



Two classes of points

Here, we're like to seperate the point (1,1) from the other points. You can see that there exists a line that does this. In fact, there exist infinite such lines. So, these two "classes" of points are linearly separable. The first class consists of the point (1,1) and the other class has (0,1), (1,0) and (0,0).

Now consider this:

Linear separability

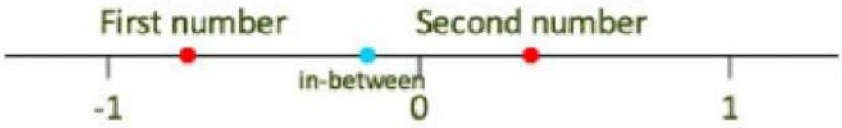
Linear separability is an important concept in neural networks. The idea is to check if you can separate points in an n-dimensional space using only n-1 dimensions. Lost it? Here's a simpler explanation.

One Dimension

Lets say you're on a number line. You take any two numbers. Now, there are two possibilities:

1. You choose two different numbers
2. You choose the same number

If you choose two different numbers, you can always find another number between them. This number "separates" the two numbers you chose.



One dimensional separability

So, you say that these two numbers are "linearly separable".

But, if both numbers are the same, you simply cannot separate them. They're the same. So, they're "linearly inseparable". (Not just linearly

- **Soma** – It is the cell body of the neuron and is responsible for processing of information, they have received from dendrites.
- **Axon** – It is just like a cable through which neurons send the information.
- **Synapses** – It is the connection between the axon and other neuron dendrites.

ANN versus BNN

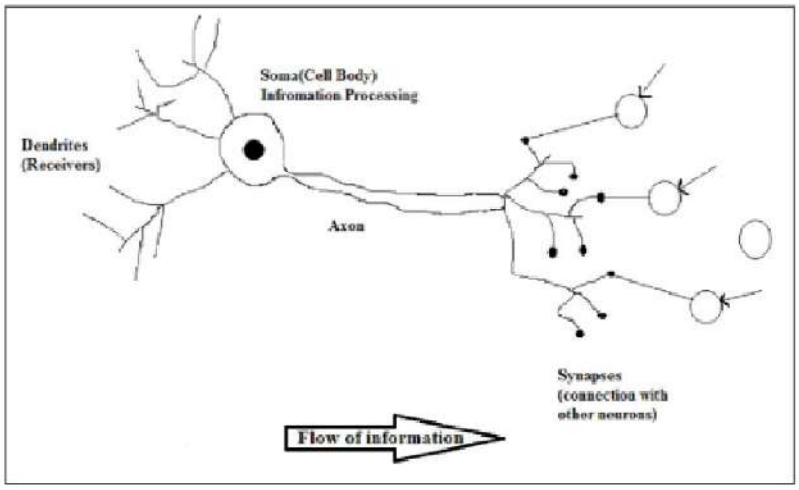
Before taking a look at the differences between Artificial Neural Network ANN and Biological Neural Network BNN, let us take a look at the similarities based on the terminology between these two.

Biological Neural Network BNN	Artificial Neural Network ANN
Soma	Node
Dendrites	Input

Biological Neuron

A nerve cell *neuron* is a special biological cell that processes information. According to an estimation, there are huge number of neurons, approximately 10^{11} with numerous interconnections, approximately 10^{15}.

Schematic Diagram



Working of a Biological Neuron

As shown in the above diagram, a typical neuron consists of the following four parts with the help of which we can explain its working −

- **Dendrites** − They are tree-like branches, responsible for receiving the information from other neurons it is connected to. In other sense, we can say that they are like the ears of neuron.