7

ARTIFICIAL NEURAL NETWORKS

Chapter Objectives

- Identify the basic components of artificial neural networks (ANNs) and their properties and capabilities.
- Describe common learning tasks such as pattern association, pattern recognition, approximation, control, and filtering that are performed by ANNs.
- Compare different ANN architecture such as feedforward and recurrent networks, and discuss their applications.
- Explain the learning process at the level of an artificial neuron, and its extension for multiplayer, feedforward-neural networks.
- Compare the learning processes and the learning tasks of competitive networks and feedforward networks.
- Present the basic principles of Kohonen maps and their applications.
- Discuss the requirements for good generalizations with ANNs based on heuristic parameter tuning.

Work on ANNs has been motivated by the recognition that the human brain computes in an entirely different way from the conventional digital computer. It was a great

Data Mining: Concepts, Models, Methods, and Algorithms, Second Edition. Mehmed Kantardzic.

challenge for many researchers in different disciplines to model the brain's computational processes. The brain is a highly complex, nonlinear, and parallel information-processing system. It has the capability to organize its components and to perform certain computations with a higher quality and many times faster than the fastest computer in existence today. Examples of these processes are pattern recognition, perception, and motor control. ANNs have been studied for more than four decades since Rosenblatt first applied the *single-layer perceptrons* to pattern-classification learning in the late 1950s.

An ANN is an abstract computational model of the human brain. The human brain has an estimated 10^{11} tiny units called neurons. These neurons are interconnected with an estimated 10^{15} links. Similar to the brain, an ANN is composed of artificial neurons (or processing units) and interconnections. When we view such a network as a graph, neurons can be represented as nodes (or vertices) and interconnections as edges. Although the term ANN is most commonly used, other names include "neural network," parallel distributed processing (PDP) system, connectionist model, and distributed adaptive system. ANNs are also referred to in the literature as neurocomputers.

A neural network, as the name indicates, is a network structure consisting of a number of nodes connected through directional links. Each node represents a processing unit, and the links between nodes specify the causal relationship between connected nodes. All nodes are adaptive, which means that the outputs of these nodes depend on modifiable parameters pertaining to these nodes. Although there are several definitions and several approaches to the ANN concept, we may accept the following definition, which views the ANN as a formalized adaptive machine:

> *An ANN is a massive parallel distributed processor made up of simple processing units. It has the ability to learn experiential knowledge expressed through interunit connection strengths, and can make such knowledge available for use.*

It is apparent that an ANN derives its computing power through, first, its massive parallel distributed structure and, second, its ability to learn and therefore to generalize. Generalization refers to the ANN producing reasonable outputs for new inputs not encountered during a learning process. The use of ANNs offers several useful properties and capabilities:

1. *Nonlinearity.* An artificial neuron as a basic unit can be a linear-or nonlinear-processing element, but the entire ANN is highly nonlinear. It is a special kind of nonlinearity in the sense that it is distributed throughout the network. This characteristic is especially important, for ANN models the inherently nonlinear real-world mechanisms responsible for generating data for learning.
2. *Learning from Examples.* An ANN modifies its interconnection weights by applying a set of training or learning samples. The final effects of a learning process are tuned parameters of a network (the parameters are distributed through the main components of the established model), and they represent implicitly stored knowledge for the problem at hand.
3. *Adaptivity.* An ANN has a built-in capability to adapt its interconnection weights to changes in the surrounding environment. In particular, an ANN

trained to operate in a specific environment can be easily retrained to deal with changes in its environmental conditions. Moreover, when it is operating in a nonstationary environment, an ANN can be designed to adopt its parameters in real time.

4. *Evidential Response.* In the context of data classification, an ANN can be designed to provide information not only about which particular class to select for a given sample, but also about confidence in the decision made. This latter information may be used to reject ambiguous data, should they arise, and thereby improve the classification performance or performances of the other tasks modeled by the network.
5. *Fault Tolerance.* An ANN has the potential to be inherently fault-tolerant, or capable of robust computation. Its performances do not degrade significantly under adverse operating conditions such as disconnection of neurons, and noisy or missing data. There is some empirical evidence for robust computation, but usually it is uncontrolled.
6. *Uniformity of Analysis and Design.* Basically, ANNs enjoy universality as information processors. The same principles, notation, and steps in methodology are used in all domains involving application of ANNs.

To explain a classification of different types of ANNs and their basic principles it is necessary to introduce an elementary component of every ANN. This simple processing unit is called an artificial neuron.

7.1 MODEL OF AN ARTIFICIAL NEURON

An artificial neuron is an information-processing unit that is fundamental to the operation of an ANN. The block diagram (Fig. 7.1), which is a model of an artificial neuron, shows that it consists of three basic elements:

1. *A set of connecting links* from different inputs x_i (or synapses), each of which is characterized by a weight or strength w_{ki}. The first index refers to the neuron in question and the second index refers to the input of the synapse to which the weight refers. In general, the weights of an artificial neuron may lie in a range that includes negative as well as positive values.
2. *An adder* for summing the input signals x_i weighted by the respective synaptic strengths w_{ki}. The operation described here constitutes a linear combiner.
3. *An activation function* f for limiting the amplitude of the output y_k of a neuron.

The model of the neuron given in Figure 7.1 also includes an externally applied bias, denoted by b_k. The bias has the effect of increasing or lowering the net input of the activation function, depending on whether it is positive or negative.

In mathematical terms, an artificial neuron is an abstract model of a natural neuron, and its processing capabilities are formalized using the following notation. First, there

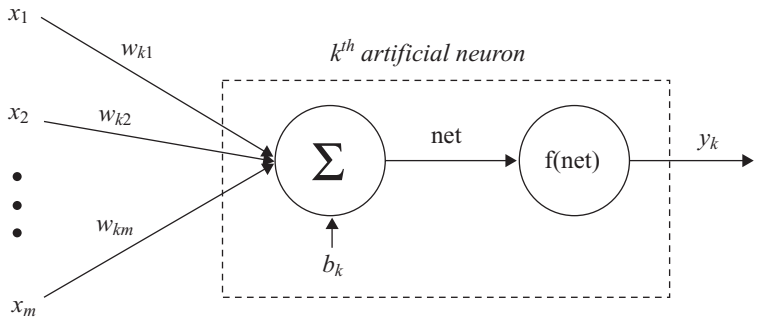


Figure 7.1. Model of an artificial neuron.

are several inputs x_i, i = 1, . . . , m. Each input x_i is multiplied by the corresponding weight w_{ki} where k is the index of a given neuron in an ANN. The weights simulate the biological synaptic strengths in a natural neuron. The weighted sum of products x_i w_{ki} for i = 1, . . . , m is usually denoted as *net* in the ANN literature:

$$net_k = x_1\, w_{k1} + x_2\, w_{k2} + \ldots + x_m\, w_{km} + b_k$$

Using adopted notation for $w_{k0} = b_k$ and default input $x_0 = 1$, a new uniform version of net summation will be

$$net_k = x_0\, w_{k0} + x_1\, w_{k1} + x_2\, w_{k2} + \ldots + x_m\, w_{km} = \sum_{i=0}^{m} x_i w_{ki}$$

The same sum can be expressed in vector notation as a scalar product of two m-dimensional vectors:

$$net_k = X \cdot W$$

where

$X = \{x_0, x_1, x_2, \ldots, x_m\}$
$W = \{w_{k0}, w_{k1}, w_{k2}, \ldots, w_{km}\}$

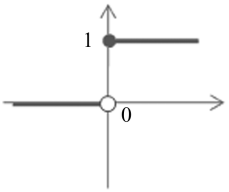
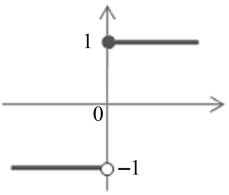
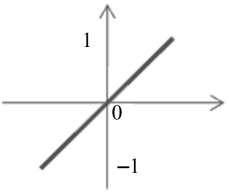
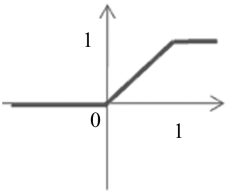
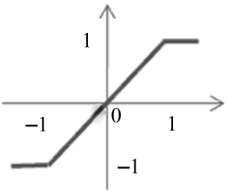
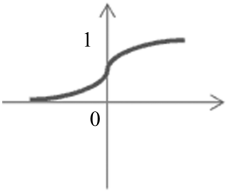
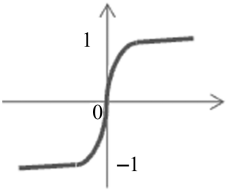
Finally, an artificial neuron computes the output y_k as a certain function of net_k value:

$$y_k = f(net_k)$$

The function f is called the activation function. Various forms of activation functions can be defined. Some commonly used activation functions are given in Table 7.1.

Now, when we introduce the basic components of an artificial neuron and its functionality, we can analyze all the processing phases in a single neuron. For example, for the neuron with three inputs and one output, the corresponding input values, weight factors, and bias are given in Figure 7.2a. It is necessary to find the output y for different activation functions such as symmetrical hard limit, saturating linear, and log-sigmoid.

TABLE 7.1. A Neuron's Common Activation Functions

Activation Function	Input/Output Relation	Graph
Hard limit	$y = \begin{cases} 1 & if\ net \geq 0 \\ 0 & if\ net < 0 \end{cases}$	
Symmetrical hard limit	$y = \begin{cases} 1 & if\ net \geq 0 \\ -1 & if\ net < 0 \end{cases}$	
Linear	$y = net$	
Saturating linear	$y = \begin{cases} 1 & if\ net > 1 \\ net & if\ 0 \leq net \leq 1 \\ 0 & if\ net < 0 \end{cases}$	
Symmetric saturating linear	$y = \begin{cases} 1 & if\ net > 1 \\ net & if\ -1 \leq net \leq 1 \\ -1 & if\ net < -1 \end{cases}$	
Log-sigmoid	$y = \frac{1}{1+e^{-net}}$	
Hyperbolic tangent sigmoid	$y = \frac{e^{net} - e^{-net}}{e^{net} + e^{-net}}$	

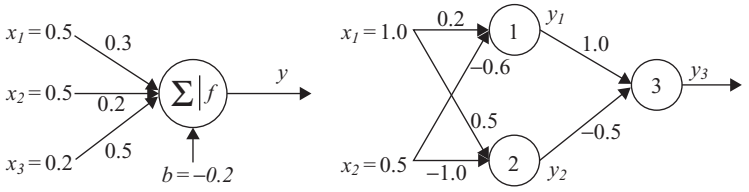


Figure 7.2. Examples of artificial neurons and their interconnections. (a) A single node; (b) three interconnected nodes.

1. Symmetrical hard limit

$$\text{net} = 0.5 \cdot 0.3 + 0.5 \cdot 0.2 + 0.2 \cdot 0.5 + (-0.2) \cdot 1 = 0.15$$

$$\text{y} = \text{f}(\text{net}) = \text{f}(0.15) = 1$$

2. Saturating linear

$$\text{net} = 0.15 \text{ (computation is the same as for case 1)}$$

$$\text{y} = \text{f}(\text{net}) = \text{f}(0.15) = 0.15$$

3. Log-sigmoid

$$\text{net} = 0.15 \text{ (computation is the same as for case 1)}$$

$$\text{y} = \text{f}(\text{net}) = \text{f}(0.15) = 1/(1 + \text{e}^{-0.15}) = 0.54$$

The basic principles of computation for one node may be extended for an ANN with several nodes even if they are in different layers, as given in Figure 7.2b. Suppose that for the given configuration of three nodes all bias values are equal to 0 and activation functions for all nodes are symmetric saturating linear. What is the final output y_3 from the node 3?

The processing of input data is layered. In the first step, the neural network performs the computation for nodes 1 and 2 that are in the first layer:

$$\text{net}_1 = 1\,0.2 + 0.5\,0.5 = 0.45 \Rightarrow \text{y}_1 = \text{f}(0.45) = 0.45$$

$$\text{net}_2 = 1\,(-0.6) + 0.5\,(-1) = -1.1 \Rightarrow \text{y}_2 = \text{f}(-1.1) = -1$$

Outputs y_1 and y_2 from the first-layer nodes are inputs for node 3 in the second layer:

$$\text{net}_3 = \text{y}_1 \cdot 1 + \text{y}_2 \cdot (-0.5) = 0.45 \cdot 1 + (-1) \cdot (-0.5) = 0.95 \Rightarrow \text{y}_3 = \text{f}(0.95) = 0.95$$

As we can see from the previous examples, the processing steps at the node level are very simple. In highly connected networks of artificial neurons, computational tasks are multiplied with an increase in the number of nodes. The complexity of processing depends on the ANN architecture.

7.2 ARCHITECTURES OF ANNS

The architecture of an ANN is defined by the characteristics of a node and the characteristics of the node's connectivity in the network. The basic characteristics of a single node have been given in a previous section and in this section the parameters of connectivity will be introduced. Typically, network architecture is specified by the number of inputs to the network, the number of outputs, the total number of elementary nodes that are usually equal processing elements for the entire network, and their organization and interconnections. Neural networks are generally classified into two categories on the basis of the type of interconnections: *feedforward* and *recurrent*.

The network is *feedforward* if the processing propagates from the input side to the output side unanimously, without any loops or feedbacks. In a layered representation of the feedforward neural network, there are no links between nodes in the same layer; outputs of nodes in a specific layer are always connected as inputs to nodes in succeeding layers. This representation is preferred because of its modularity, that is, nodes in the same layer have the same functionality or generate the same level of abstraction about input vectors. If there is a feedback link that forms a circular path in a network (usually with a delay element as a synchronization component), then the network is *recurrent*. Examples of ANNs belonging to both classes are given in Figure 7.3.

Although many neural-network models have been proposed in both classes, the multilayer feedforward network with a backpropagation-learning mechanism is the most widely used model in terms of practical applications. Probably over 90% of

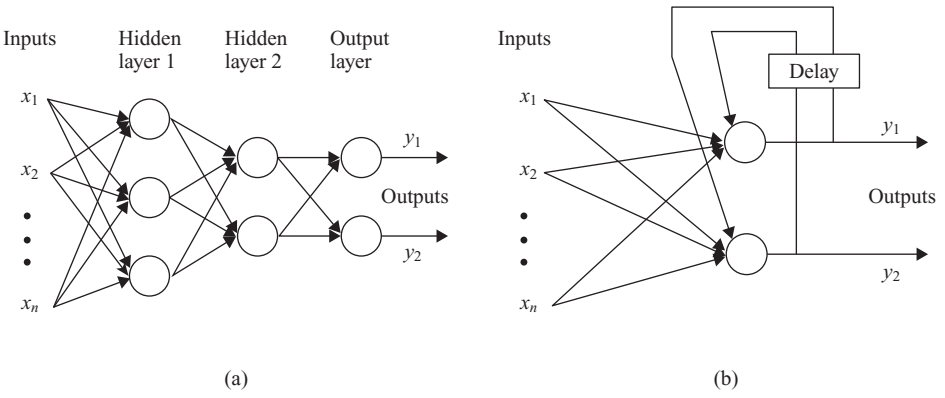


Figure 7.3. Typical architectures of artificial neural networks. (a) Feedforward network; (b) recurrent network.

commercial and industrial applications are based on this model. Why multilayered networks? A simple example will show the basic differences in application requirements between single-layer and multilayer networks.

The simplest and well-known classification problem, very often used as an illustration in the neural-network literature, is the exclusive-OR (XOR) problem. The task is to classify a binary input vector X to class 0 if the vector has an even number of 1's or otherwise assign it to class 1. The XOR problem is not linearly separable; this can easily be observed from the plot in Figure 7.4 for a two-dimensional (2-D) input vector $X = \{x_1, x_2\}$. There is no possibility of obtaining a single linear separation of points that belong to different classes. In other words, we cannot use a single-layer network to construct a straight line (in general, it is a linear hyperplane in an n-dimensional space) to partition the 2-D input space into two regions, each containing data points of only the same class. It is possible to solve the problem with a two-layer network, as illustrated in Figure 7.5, in which one possible solution for the connection weights and

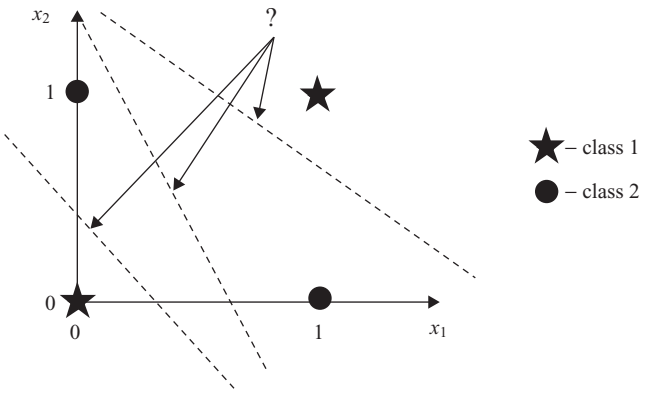


Figure 7.4. XOR problem.

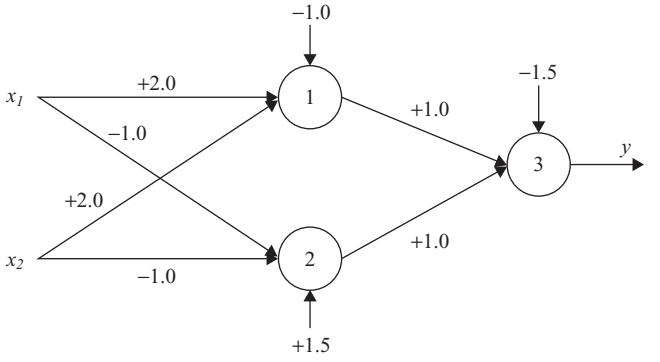


Figure 7.5. XOR solution: the two-layer ANN with the hard-limit activation function.

thresholds is indicated. This network generates a nonlinear separation of points in a 2-D space.

The basic conclusion from this example is that single-layered ANNs are a convenient modeling tool only for relatively simple problems that are based on linear models. For most real-world problems, where models are highly nonlinear, multilayered networks are better and maybe the only solution.

7.3 LEARNING PROCESS

A major task for an ANN is to learn a model of the world (environment) in which it is embedded and to maintain the model sufficiently consistent with the real world so as to achieve the specified goals of the concerned application. The learning process is based on data samples from the real world, and here lies a fundamental difference between the design of an ANN and a classical information-processing system. In the latter case, we usually proceed by first formulating a mathematical model of environmental observations, validating the model with real data, and then building (programming) the system on the basis of the model. In contrast, the design of an ANN is based directly on real-life data, with the data set being permitted to "speak for itself." Thus, an ANN not only provides the implicit model formed through the learning process, but also performs the information-processing function of interest.

The property that is of primary significance for an ANN is the ability of the network to learn from its environment based on real-life examples, and to improve its performance through that learning process. An ANN learns about its environment through an interactive process of adjustments applied to its connection weights. Ideally, the network becomes more knowledgeable about its environment after each iteration in the learning process. It is very difficult to agree on a precise definition of the term learning. In the context of ANNs one possible definition of inductive learning is as follows:

> *Learning is a process by which the free parameters of a neural network are adapted through a process of stimulation by the environment in which the network is embedded. The type of learning is determined by the manner in which the parameters change.*

A prescribed set of well-defined rules for the solution of a learning problem is called a learning algorithm. Basically, learning algorithms differ from each other in the way in which the adjustment of the weights is formulated. Another factor to be considered in the learning process is the manner in which ANN architecture (nodes and connections) is built.

To illustrate one of the learning rules, consider the simple case of a neuron k, shown in Figure 7.1, constituting the only computational node of the network. Neuron k is driven by input vector $X(n)$, where n denotes discrete time, or, more precisely, the time step of the iterative process involved in adjusting the input weights w_{ki}. Every data sample for ANN training (learning) consists of the input vector $X(n)$ and the corresponding output $d(n)$.

	Inputs	Output
Sample_k	$x_{k1}, x_{k2}, \ldots, x_{km}$	d_k

Processing the input vector $X(n)$, a neuron k produces the output that is denoted by $y_k(n)$:

$$y_k = f\left(\sum_{i=1}^{m} x_i \, w_{ki}\right)$$

It represents the only output of this simple network, and it is compared with a desired response or target output $d_k(n)$ given in the sample. An error $e_k(n)$ produced at the output is by definition

$$e_k(n) = d_k(n) - y_k(n)$$

The error signal produced actuates a control mechanism of the learning algorithm, the purpose of which is to apply a sequence of corrective adjustments to the input weights of a neuron. The corrective adjustments are designed to make the output signal $y_k(n)$ come closer to the desired response $d_k(n)$ in a step-by-step manner. This objective is achieved by minimizing a cost function $E(n)$, which is the instantaneous value of error energy, defined for this simple example in terms of the error $e_k(n)$:

$$E(n) = \tfrac{1}{2} e_k^2(n)$$

The learning process based on a minimization of the cost function is referred to as *error-correction learning*. In particular, minimization of $E(n)$ leads to a learning rule commonly referred to as the *delta rule* or Widrow-Hoff rule. Let $w_{kj}(n)$ denote the value of the weight factor for neuron k excited by input $x_j(n)$ at time step n. According to the delta rule, the adjustment $\Delta w_{kj}(n)$ is defined by

$$\Delta w_{kj}(n) = \eta \cdot e_k(n) \, x_j(n)$$

where η is a positive constant that determines the rate of learning. Therefore, the delta rule may be stated as: The adjustment made to a weight factor of an input neuron connection is proportional to the product of the error signal and the input value of the connection in question.

Having computed the adjustment $\Delta w_{kj}(n)$, the updated value of synaptic weight is determined by

$$w_{kj}(n+1) = w_{kj}(n) + \Delta w_{kj}(n)$$

In effect, $w_{kj}(n)$ and $w_{kj}(n + 1)$ may be viewed as the old and new values of synaptic weight w_{kj}, respectively. From Figure 7.6 we recognize that error-correction learning is an example of a closed-loop feedback system. Control theory explains that the stability of such a system is determined by those parameters that constitute the feedback

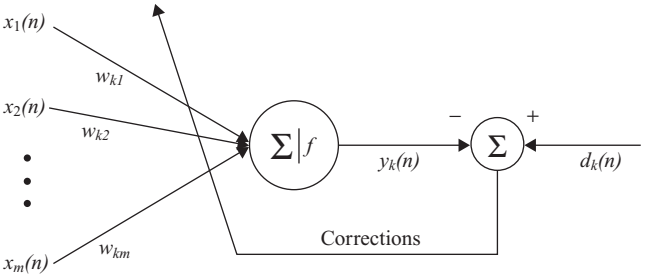
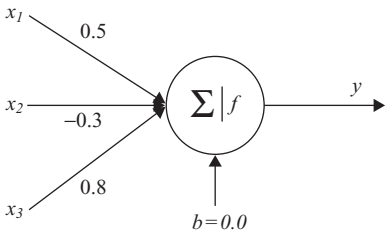


Figure 7.6. Error-correction learning performed through weights adjustments.



(a)

n (sample)	x_1	x_2	x_3	d
1	1.0	1	0.5	0.7
2	−1.0	0.7	−0.5	0.2
3	0.3	0.3	−0.3	0.5

(b)

Figure 7.7. Initialization of the error correction-learning process for a single neuron. (a) Artificial neuron with the feedback; (b) training data set for a learning process.

loop. One of those parameters of particular interest is the learning rate η. This parameter has to be carefully selected to ensure that the stability of convergence of the iterative-learning process is achieved. Therefore, in practice, this parameter plays a key role in determining the performance of error-correction learning.

Let us analyze one simple example of the learning process performed on a single artificial neuron in Figure 7.7a, with a set of the three training (or learning) examples given in Figure 7.7b.

The process of adjusting the weight factors for a given neuron will be performed with the learning rate $\eta = 0.1$. The bias value for the neuron is equal 0, and the activation function is linear. The first iteration of a learning process, and only for the first training example, is performed with the following steps:

$$\text{net}(1) = 0.5 \cdot 1 + (-0.3) \cdot 1 + 0.8 \cdot 0.5 = 0.6$$

$$\Downarrow$$

$$\text{y}(1) = \text{f}(\text{net}(1)) = \text{f}(0.6)) = 0.6$$

$$\Downarrow$$

$$\text{e}(1) = \text{d}(1) - \text{y}(1) = 0.7 - 0.6 = 0.1$$

$$\Downarrow$$

$$\Delta \text{w}_1(1) = 0.1 \cdot 0.1 \cdot 1 = 0.01 \Rightarrow \text{w}_1(2) = \text{w}_1(1) + \Delta \text{w}_1(1) = 0.5 + 0.01 = 0.51$$

TABLE 7.2. Adjustment of Weight Factors with Training Examples in Figure 7.7b

Parameter	$n = 2$	$n = 3$
x_1	–1	0.3
x_2	0.7	0.3
x_3	–0.5	–0.3
y	–1.1555	–0.18
d	0.2	0.5
e	1.3555	0.68
$\Delta w_1(n)$	–0.14	0.02
$\Delta w_2(n)$	0.098	0.02
$\Delta w_3(n)$	–0.07	–0.02
$w1(n + 1)$	0.37	0.39
$w2(n + 1)$	–0.19	–0.17
$w3(n + 1)$	0.735	0.715

$$\Delta w_2(1) = 0.1 \cdot 0.1 \cdot 1 = 0.01 \Rightarrow w_2(2) = w_2(1) + \Delta w_2(1) = -0.3 + 0.01 = -0.29$$

$$\Delta w_3(1) = 0.1 \cdot 0.1 \cdot 0.5 = 0.005 \Rightarrow w_3(2) = w_3(1) + \Delta w_3(1) = 0.8 + 0.005 = 0.805$$

Similarly, it is possible to continue with the second and third examples ($n = 2$ and $n = 3$). The results of the learning corrections Δw together with new weight factors w are given in Table 7.2.

Error-correction learning can be applied on much more complex ANN architecture, and its implementation is discussed in Section 7.5, where the basic principles of multilayer feedforward ANNs with backpropagation are introduced. This example only shows how weight factors change with every training (learning) sample. We gave the results only for the first iteration. The weight-correction process will continue either by using new training samples or by using the same data samples in the next iterations. As to when to finish the iterative process is defined by a special parameter or set of parameters called *stopping criteria*. A learning algorithm may have different stopping criteria, such as the maximum number of iterations, or the threshold level of the weight factor may change in two consecutive iterations. This parameter of learning is very important for final learning results and it will be discussed in later sections.

7.4 LEARNING TASKS USING ANNS

The choice of a particular learning algorithm is influenced by the learning task that an ANN is required to perform. We identify six basic learning tasks that apply to the use of different ANNs. These tasks are subtypes of general learning tasks introduced in Chapter 4.

7.4.1 Pattern Association

Association has been known to be a prominent feature of human memory since Aristotle, and all models of cognition use association in one form or another as the basic operation. Association takes one of two forms: *autoassociation* or *heteroassociation*. In *autoassociation*, an ANN is required to store a set of patterns by repeatedly presenting them to the network. The network is subsequently presented with a partial description or a distorted, noisy version of an original pattern, and the task is to retrieve and recall that particular pattern. *Heteroassociation* differs from *autoassociation* in that an arbitrary set of input patterns is paired with another arbitrary set of output patterns. *Autoassociation* involves the use of unsupervised learning, whereas *heteroassociation* learning is supervised. For both, *autoassociation* and *heteroassociation*, there are two main phases in the application of an ANN for pattern-association problems:

1. the storage phase, which refers to the training of the network in accordance with given patterns, and
2. the recall phase, which involves the retrieval of a memorized pattern in response to the presentation of a noisy or distorted version of a key pattern to the network.

7.4.2 Pattern Recognition

Pattern recognition is also a task that is performed much better by humans than by the most powerful computers. We receive data from the world around us via our senses and are able to recognize the source of the data. We are often able to do so almost immediately and with practically no effort. Humans perform pattern recognition through a learning process, so it is with ANNs.

Pattern recognition is formally defined as the process whereby a received pattern is assigned to one of a prescribed number of classes. An ANN performs pattern recognition by first undergoing a training session, during which the network is repeatedly presented a set of input patterns along with the category to which each particular pattern belongs. Later, in a testing phase, a new pattern is presented to the network that it has not seen before, but which belongs to the same population of patterns used during training. The network is able to identify the class of that particular pattern because of the information it has extracted from the training data. Graphically, patterns are represented by points in a multidimensional space. The entire space, which we call decision space, is divided into regions, each one of which is associated with a class. The decision boundaries are determined by the training process, and they are tested if a new, unclassified pattern is presented to the network. In essence, pattern recognition represents a standard classification task.

Function Approximation. Consider a nonlinear input–output mapping described by the functional relationship

$$Y = f(X)$$

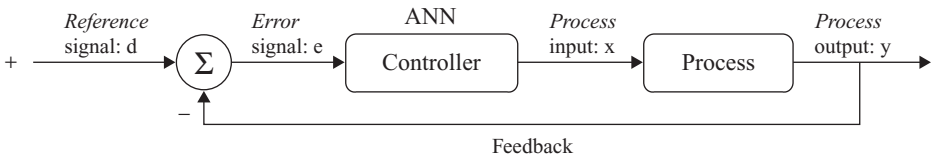


Figure 7.8. Block diagram of ANN-based feedback-control system.

where the vector X is the input and Y is the output. The vector-value function f is assumed to be unknown. We are given the set of labeled examples $\{X_i, Y_i\}$, and we have to design an ANN that approximates the unknown function f with a function F that is very close to original function. Formally:

$$|F(X_i) - f(X_i)| < \varepsilon \text{ for all } X_i \text{ from the training set}$$

where ε is a small positive number. Provided that the size of the training set is large enough and the network is equipped with an adequate number of free parameters, the approximation error ε can be made small enough for the task. The approximation problem described here is a perfect candidate for supervised learning.

Control Control is another learning task that can be done by an ANN. Control is applied to a process or a critical part in a system, which has to be maintained in a controlled condition. Consider the control system with feedback shown in Figure 7.8.

The system involves the use of feedback to control the output y on the level of a reference signal d supplied from the external source. A controller of the system can be realized in an ANN technology. The error signal e, which is the difference between the process output y and the reference value d, is applied to an ANN-based controller for the purpose of adjusting its free parameters. The primary objective of the controller is to supply appropriate inputs x to the process to make its output y track the reference signal d. It can be trained through:

1. *Indirect Learning.* Using actual input–output measurements on the process, an ANN model of a control is first constructed offline. When the training is finished, the ANN controller may be included into the real-time loop.
2. *Direct Learning.* The training phase is online, with real-time data, and the ANN controller is enabled to learn the adjustments to its free parameters directly from the process.

Filtering The term filter often refers to a device or algorithm used to extract information about a particular quantity from a set of noisy data. Working with series of data in time domain, frequent domain, or other domains, we may use an ANN as a filter to perform three basic information-processing tasks:

1. *Filtering.* This task refers to the extraction of information about a particular quantity at discrete time *n* by using data measured up to and including time *n*.

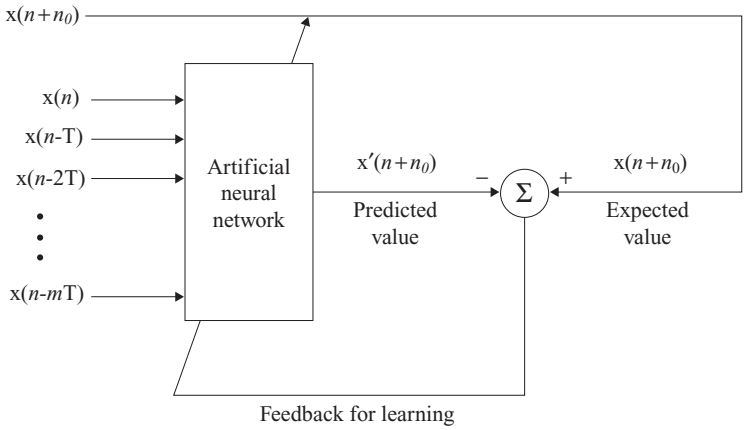


Figure 7.9. Block diagram of an ANN-based prediction.

2. *Smoothing.* This task differs from filtering in that data need not be available only at time n; data measured later than time n can also be used to obtain the required information. This means that in smoothing there is a delay in producing the result at discrete time n.
3. *Prediction.* The task of prediction is to forecast data in the future. The aim is to derive information about what the quantity of interest will be like at some time $n + n_0$ in the future, for $n_0 > 0$, by using data measured up to and including time n. Prediction may be viewed as a form of model building in the sense that the smaller we make the prediction error, the better the network serves as a model of the underlying physical process responsible for generating the data. The block diagram of an ANN for a prediction task is given in Figure 7.9.

7.5 MULTILAYER PERCEPTRONS (MLPs)

Multilayer feedforward networks are one of the most important and most popular classes of ANNs in real-world applications. Typically, the network consists of a set of inputs that constitute the input layer of the network, one or more hidden layers of computational nodes, and finally an output layer of computational nodes. The processing is in a forward direction on a layer-by-layer basis. This type of ANNs are commonly referred to as MLPs, which represent a generalization of the simple perceptron, a network with a single layer, considered earlier in this chapter.

A multiplayer perceptron has three distinctive characteristics:

1. The model of each neuron in the network includes usually a nonlinear activation function, sigmoidal or hyperbolic.
2. The network contains one or more layers of hidden neurons that are not a part of the input or output of the network. These hidden nodes enable the network

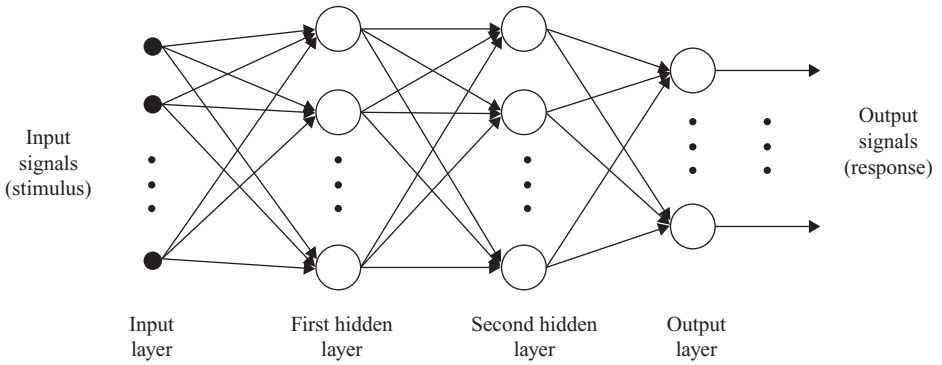


Figure 7.10. A graph of a multilayered-perceptron architecture with two hidden layers.

to learn complex and highly nonlinear tasks by extracting progressively more meaningful features from the input patterns.

3. The network exhibits a high degree of connectivity from one layer to the next one.

Figure 7.10 shows the architectural graph of a multilayered perceptron with two hidden layers of nodes for processing and an output layer. The network shown here is fully connected. This means that the neuron in any layer of the network is connected to all the nodes (neurons) in the previous layer. Data flow through the network progresses in a forward direction, from left to right and on a layer-by-layer basis.

MLPs have been applied successfully to solve some difficult and diverse problems by training the network in a supervised manner with a highly popular algorithm known as the *error backpropagation algorithm*. This algorithm is based on the error-correction learning rule and it may be viewed as its generalization. Basically, error backpropagation learning consists of two phases performed through the different layers of the network: a forward pass and a backward pass.

In the forward pass, a training sample (input data vector) is applied to the input nodes of the network, and its effect propagates through the network layer by layer. Finally, a set of outputs is produced as the actual response of the network. During the forward phase, the synaptic weights of the network are all fixed. During the backward phase, on the other hand, the weights are all adjusted in accordance with an error-correction rule. Specifically, the actual response of the network is subtracted from a desired (target) response, which is a part of the training sample, to produce an error signal. This error signal is then propagated backward through the network, against the direction of synaptic connections. The synaptic weights are adjusted to make the actual response of the network closer to the desired response.

Formalization of the backpropagation algorithm starts with the assumption that an error signal exists at the output of a neuron j at iteration n (i.e., presentation of the nth training sample). This error is defined by

$$e_j(n) = d_j(n) - y_j(n)$$

We define the instantaneous value of the error energy for neuron j as $1/2\ e_j^2(n)$. The total error energy for the entire network is obtained by summing instantaneous values over all neurons in the output layer. These are the only "visible" neurons for which the error signal can be calculated directly. We may thus write

$$E(n) = \tfrac{1}{2} \sum_{j \in C} e_j^2(n),$$

where the set C includes all neurons in the output layer of the network. Let N denote the total number of samples contained in the training set. The average squared error energy is obtained by summing E(*n*) over all *n* and then normalizing it with respect to size N, as shown by

$$E_{av} = 1/N \sum_{n=1}^{N} E(n)$$

The average error energy E_{av} is a function of all the free parameters of the network. For a given training set, E_{av} represents the cost function as a measure of learning performances. The objective of the learning process is to adjust the free parameters of the network to minimize E_{av}. To do this minimization, the weights are updated on a sample-by-sample basis for one iteration, that is, one complete presentation of the entire training set of a network has been dealt with.

To obtain the minimization of the function E_{av}, we have to use two additional relations for node-level processing, which have been explained earlier in this chapter:

$$v_j(n) = \sum_{i=1}^{m} w_{ji}(n)\, x_i(n)$$

and

$$y_j(n) = \varphi(v_j(n))$$

where m is the number of inputs for j^{th} neuron. Also, we use the symbol v as a shorthand notation of the previously defined variable *net*. The backpropagation algorithm applies a correction $\Delta w_{ji}(n)$ to the synaptic weight $w_{ji}(n)$, which is proportional to the partial derivative $\delta E(n)/\delta w_{ji}(n)$. Using the chain rule for derivation, this partial derivative can be expressed as

$$\partial E(n)/\partial w_{ji}(n) = \partial E(n)/\partial e_j(n) \cdot \partial e_j(n)/\partial y_j(n) \cdot \partial y_j(n)/\partial v_j(n) \cdot \partial v_j(n)/\partial w_{ji}(n)$$

The partial derivative $\delta E(n)/\delta wji(n)$ represents a sensitive factor, determining the direction of search in weight space. Knowing that the next relations

$$\partial E(n)/\partial e_j(n) = e_j(n) \text{ (from } E(n) = \tfrac{1}{2} \Sigma e_j^2(n))$$

$$\partial e_j(n)/\partial y_j(n) = -1 \text{ (from } e_j(n) = d_j(n) - y_j(n))$$

$$\partial y_j(n)/\partial v_j(n) = \varphi'(v_j(n)) \text{ (from } y_j(n) = \varphi[v_j(n)])$$

$$\partial v_j(n)/\partial w_{ji}(n) = x_i(n) \text{ (from } \Sigma w_{ji}(n)\, x_i(n))$$

are valid, we can express the partial derivative $\partial E(n)/\partial w_{ji}(n)$ in the form

$$\partial E(n)/\partial w_{ji}(n) = -e_j(n)\varphi'(v_j(n))\, x_i(n)$$

The correction $\Delta w_{ji}(n)$ applied to $w_{ji}(n)$ is defined by the delta rule

$$\Delta w_{ji}(n) = -\eta \cdot \partial E(n)/\partial w_{ji}(n) = \eta \cdot e_j(n) \cdot \varphi'(v_j(n)) \cdot x_i(n)$$

where η is the learning-rate parameter of the backpropagation algorithm. The use of the minus sign accounts for gradient descent in weight space, that is, a direction for weight change that reduces the value $E(n)$. Asking for $\varphi'(v_j[n])$ in the learning process is the best explanation for why we prefer continuous functions such as log-sigmoid and hyperbolic as a standard-activation function at the node level. Using the notation $\delta_j(n) = e_j(n)\cdot\varphi'_j(v_j[n])$, where $\delta_j(n)$ is the *local gradient*, the final equation for $w_{ji}(n)$ corrections is

$$\Delta w_{ji}(n) = \eta \cdot \delta_j(n) \cdot x_i(n)$$

The local gradient $\delta_j(n)$ points to the required changes in synaptic weights. According to its definition, the local gradient $\delta_j(n)$ for output neuron j is equal to the product of the corresponding error signal $e_j(n)$ for that neuron and the derivative $\varphi'(v_j[n])$ of the associated activation function.

Derivative $\varphi'(v_j[n])$ can be easily computed for a standard activation function, where differentiation is the only requirement for the function. If the activation function is sigmoid, it means that in the form

$$y_j(n) = \varphi(v_j(n)) = 1/(1 + e^{(--vj(n))})$$

the first derivative is

$$\varphi'(v_j[n]) = e^{(-vj(n))}/(1 + e^{(-vj(n)})^2 = y_j(n)(1 - y_j(n))$$

and a final weight correction is

$$\Delta w_{ji}(n) = \eta \cdot e_j(n) \cdot y_j(n)(1 - y_j(n)) \cdot x_i(n)$$

The final correction $\Delta w_{ji}(n)$ is proportional to the learning rate η, the error value at this node is $e_j(n)$, and the corresponding input and output values are $x_i(n)$ and $y_j(n)$. Therefore, the process of computation for a given sample n is relatively simple and straightforward.

If the activation function is a hyperbolic tangent, a similar computation will give the final value for the first derivative $\varphi'(v_j[n])$:

$$\varphi'(v_j[n]) = (1 - y_j[n]) \cdot (1 + y_j[n])$$

and

$$\Delta w_{ji}(n) = \eta \cdot e_j(n) \cdot (1 - y_j[n]) \cdot (1 + y_j[n]) \cdot x_i(n)$$

Again, the practical computation of $\Delta w_{ji}(n)$ is very simple because the local-gradient derivatives depend only on the output value of the node $y_j(n)$.

In general, we may identify two different cases of computation for $\Delta w_{ji}(n)$, depending on where in the network neuron j is located. In the first case, neuron j is an output node. This case is simple to handle because each output node of the network is supplied with a desired response, making it a straightforward matter to calculate the associated error signal. All previously developed relations are valid for output nodes without any modifications.

In the second case, neuron j is a hidden node. Even though hidden neurons are not directly accessible, they share responsibility for any error made at the output of the network. We may redefine the local gradient $\delta_j(n)$ for a hidden neuron j as the product of the associated derivative $\varphi'(v_j[n])$ and the weighted sum of the local gradients computed for the neurons in the next layer (hidden or output) that are connected to neuron j

$$\delta_j(n) = \varphi'(v_j(n)) \sum_k \delta_k(n) \cdot w_{kj}(n), \qquad k \in D$$

where D denotes the set of all nodes on the next layer that are connected to the node j. Going backward, all $\delta_k(n)$ for the nodes in the next layer are known before computation of the local gradient $\delta_j(n)$ for a given node on a layer closer to the inputs.

Let us analyze once more the application of the backpropagation-learning algorithm with two distinct passes of computation that are distinguished for each training example. In the first pass, which is referred to as the forward pass, the function signals of the network are computed on a neuron-by-neuron basis, starting with the nodes on first hidden layer (the input layer is without computational nodes), then the second, and so on, until the computation is finished with final output layer of nodes. In this pass, based on given input values of each learning sample, a network computes the corresponding output. Synaptic weights remain unaltered during this pass.

The second, backward pass, on the other hand, starts at the output layer, passing the error signal (the difference between the computed and the desired output value) leftward through the network, layer by layer, and recursively computing the local gradients δ for each neuron. This recursive process permits the synaptic weights of the network to undergo changes in accordance with the delta rule. For the neuron located at the output layer, δ is equal to the error signal of that neuron multiplied by the first derivative of its nonlinearity represented in the activation function. Based on local gradients δ, it is straightforward to compute Δw for each connection to the output nodes. Given the δ values for all neurons in the output layer, we use them in the previous layer before (usually the hidden layer) to compute modified local gradients for the nodes that are not the final, and again to correct Δw for input connections for this layer. The backward procedure is repeated until all layers are covered and all weight factors in

the network are modified. Then, the backpropagation algorithm continues with a new training sample. When there are no more training samples, the first iteration of the learning process finishes. With the same samples, it is possible to go through a second, third, and sometimes hundreds of iterations until error energy E_{av} for the given iteration is small enough to stop the algorithm.

The backpropagation algorithm provides an "approximation" to the trajectory in weight space computed by the method of steepest descent. The smaller we make the learning rate parameter η, the smaller the changes to the synaptic weights in the network will be from one iteration to the next and the smoother will be the trajectory in weight space. This improvement, however, is attained at the cost of a slower rate of learning. If, on the other hand, we make η too large in order to speed up the learning process, the resulting large changes in the synaptic weights can cause the network to become unstable, and the solution will become oscillatory about a minimal point never reaching it.

A simple method of increasing the rate of learning yet avoiding the danger of instability is to modify the delta rule by including a *momentum term*:

$$\Delta w_{ji}(n) = \eta \cdot \delta_j(n) \cdot x_i(n) + \alpha \cdot \Delta w_{ji}(n-1)$$

where α is usually a positive number called momentum constant and $\Delta w_{ji}(n-1)$ is the correction of the weight factor for a previous $(n-1)^{th}$ sample. α, in practice, is usually set to the value between 0.1 and 1. The addition of the momentum term smoothes the weight updating and tends to resist erratic weight changes because of 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. The momentum factor represents a method of averaging; rather than averaging derivatives, momentum averages the weight changes themselves. The idea behind momentum is apparent from its name: including some kind of inertia in weight corrections. The inclusion of the momentum term in the backpropagation algorithm has a stabilizing effect in cases where corrections in weight factors have a high oscillation and sign changes. The momentum term may also have the benefit of preventing the learning process from terminating in a shallow local minimum on the error surface.

Reflecting practical approaches to the problem of determining the optimal architecture of the network for a given task, the question about the values for three parameters, the number of hidden nodes (including the number of hidden layers), learning rate η, and momentum rate α, becomes very important. Usually the optimal architecture is determined experimentally, but some practical guidelines exist. If several networks with different numbers of hidden nodes give close results with respect to error criteria after the training, then the best network architecture is the one with smallest number of hidden nodes. Practically, that means starting the training process with networks that have a small number of hidden nodes, increasing this number, and then analyzing the resulting error in each case. If the error does not improve with the increasing number of hidden nodes, the latest analyzed network configuration can be selected as optimal. Optimal learning and momentum constants are also determined experimentally, but experience shows that the solution should be found with η about 0.1 and α about 0.5.

When the ANN is first set up, the initial weight factors must be given. The goal in choosing these values is to begin the learning process as fast as possible. The appropriate method is to take the initial weights as very small evenly distributed random numbers. That will cause the output values to be in mid-range regardless of the values of its inputs, and the learning process will converge much faster with every new iteration.

In backpropagation learning, we typically use the algorithm to compute the synaptic weights by using as many training samples as possible. The hope is that the neural network so designed will generalize the best. A network is said to generalize well when the input–output mapping computed by the network is correct for test data never used earlier in creating or training the network. In the MLP, if the number of hidden units is less that the number of inputs, the first layer performs a dimensionality reduction. Each hidden unit may be interpreted as defining a template. By analyzing these templates we can extract knowledge from a trained ANN. In this interpretation weights are defining relative importance in the templates. But the largest number of training samples and the largest number of learning iterations using these samples do not necessarily lead to the best generalization. Additional problems occur during the learning process, and they are briefly described through the following analysis.

The learning process using an ANN may be viewed as a curve-fitting problem. Such a viewpoint then permits us to look on generalization not as a theoretical property of neural networks but as the effect of a good, nonlinear interpolation of the input data. An ANN that is designed to generalize well will produce a correct input–output mapping, even when the input is slightly different from the samples used to train the network, as illustrated in Figure 7.11a. When, however, an ANN learns from too many input–output samples, the network may end up memorizing the training data. Such a phenomenon is referred to as *overfitting* or *overtraining*. This problem has already been described in Chapter 4. When the network is overtrained, it loses the ability to generalize between similar patterns. A smoothness of input–output mapping, on the other hand,

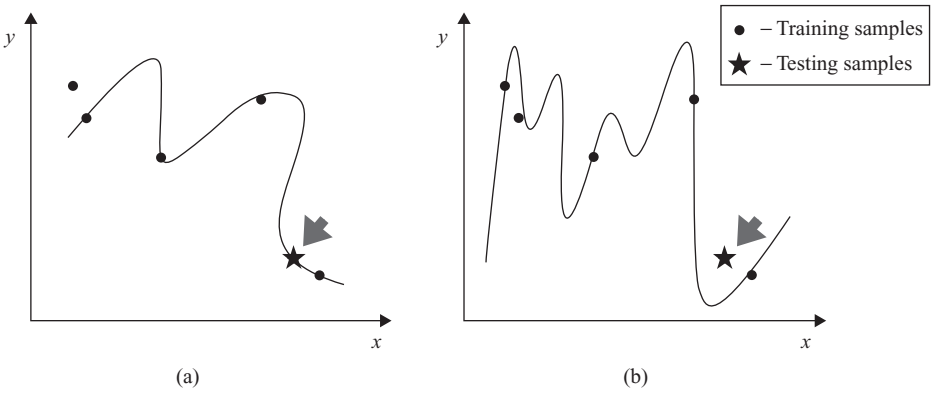


Figure 7.11. Generalization as a problem of curve fitting. (a) A fitting curve with good generalization; (b) overfitted curve.

is closely related to the generalization abilities of an ANN. The essence is to select, based on training data, the simplest function for generalization, that means the smoothest function that approximates the mapping for a given error criterion. Smoothness is natural in many applications, depending on the scale of the phenomenon being studied. It is therefore important to seek a smooth nonlinear mapping, so that the network is able to classify novel patterns correctly with respect to the training patterns. In Figure 7.11a,b, a fitting curve with a good generalization and an overfitted curve are represented for the same set of training data.

To overcome the problem of overfitting, some additional practical recommendations may be introduced for the design and application of ANN in general and multiplayer perceptrons in particular. In ANNs, as in all modeling problems, we want to use the simplest network that can adequately represent the training data set. Do not use a bigger network when a smaller network will work. An alternative to using the simplest network is to stop the training before the network overfits. Also, one very important constraint is that the number of network parameters should be limited. For a network to be able to generalize it should have fewer parameters (significantly) than there are data points in the training set. ANN generalization is extremely poor if there is a large input space with very few training samples.

Interpretability of data-mining models including ANNs, or the understanding of the way inputs relate to an output in a model, is a desirable property in applied data-mining research because the intent of such studies is to gain knowledge about the underlying reasoning mechanisms. Interpretation may also be used to validate results that are inconsistent or contradictory to common understanding of issues involved, and it may also indicate problems with data or models.

While ANNs have been intensively studied and successfully used in classification and regression problems, their interpretability still remains vague. They suffer from the shortcoming of being "black boxes," that is, without explicit mechanisms for determining why an ANN makes a particular decision. That is, one provides the input values for an ANN and obtains an output value, but generally no information is provided regarding how those outputs were obtained, how the input values correlate to the output value, and what is the meaning of large numbers of weight factors in the network. ANNs acceptability as valid data-mining methods for business and research requires that beyond providing excellent predictions they provide meaningful insight that can be understood by a variety of users: clinicians, policy makers, business planners, academicians, and lay persons. Human understanding and acceptance is greatly enhanced if the input–output relations are explicit, and end users would gain more confidence in the prediction produced.

Interpretation of trained ANNs can be considered in two forms: broad and detailed. The aim of a broad interpretation is to characterize how important an input neuron is for predictive ability of the model. This type of interpretation allows us to rank input features in order of importance. The broad interpretation is essentially a sensitivity analysis of the neural network. The methodology does not indicate the sign or direction of the effect of each input. Thus, we cannot draw conclusions regarding the nature of the correlation between input descriptors and network output; we are only concluding about the level of influence.

The goal of a detailed interpretation of an ANN is to extract the structure-property trends from an ANN model. For example, each of the hidden neurons corresponds to the number of piecewise hyperplanes that are components available for approximating the target function. These hyperplanes act as the basic building blocks for constructing an explicit ANN model. To obtain a more comprehensible system that approximates the behavior of the ANN, we require the model with less complexity, and at the same time maybe scarifying accuracy of results. The knowledge hidden in a complex structure of an ANN may be uncovered using a variety of methodologies that allow mapping an ANN into a rule-based system. Many authors have focused their activities on compiling the knowledge captured in the topology and weight matrix of a neural network into a symbolic form: some of them into sets of ordinary if-then rules, others into formulas from propositional logic or from non-monotonic logics, or most often into sets of fuzzy rules. These transformations make explicit the knowledge implicitly captured by the trained neural network and it allows the human specialist to understand how the neural network generates a particular result. It is important to emphasize that any method of rule extraction from ANN is valuable only to the degree to which the extracted rules are meaningful and comprehensible to a human expert.

It is proven that the best interpretation of trained ANNs with continuous activation functions is in a form of fuzzy rule-based systems. In this way, a more comprehensible description of the action of the ANN is achieved. Multilayer feedforward ANNs are seen as additive fuzzy rule-based systems. In these systems, the outputs of each rule are weighted by the activation degree of the rule, and then they are added for an integrated representation of an ANN model. The main disadvantage of most approximation techniques of neural networks by fuzzy rules is the exponential increase of required number of rules for a good approximation. Fuzzy rules that express the input–output mapping of the ANNs are extracted using different approaches described in numerous references. If the reader is interested for more details about methodologies, the starting points may be the recommended references at the end of this chapter, and also the introductory concepts about fuzzy systems given in Chapter 14.

7.6 COMPETITIVE NETWORKS AND COMPETITIVE LEARNING

Competitive neural networks belong to a class of recurrent networks, and they are based on algorithms of unsupervised learning, such as the competitive algorithm explained in this section. In competitive learning, the output neurons of a neural network compete among themselves to become active (to be "fired"). Whereas in multiplayer perceptrons several output neurons may be active simultaneously, in competitive learning only a single output neuron is active at any one time. There are three basic elements necessary to build a network with a competitive learning rule, a standard technique for this type of ANNs:

1. a *set of neurons* that have the same structure and that are connected with initially randomly selected weights; therefore, the neurons respond differently to a given set of input samples;

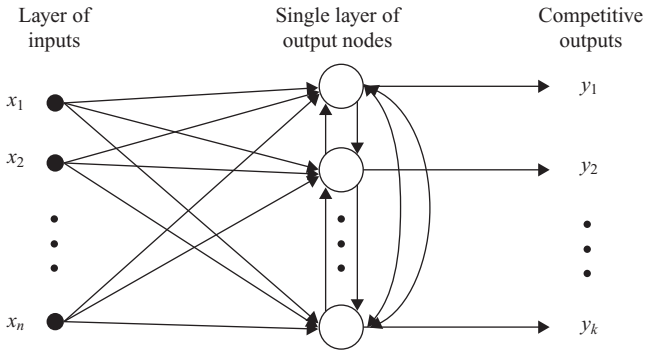


Figure 7.12. A graph of a simple competitive network architecture.

2. a *limit value* that is determined on the strength of each neuron; and
3. a *mechanism that permits the neurons to compete* for the right to respond to a given subset of inputs, such that only one output neuron is active at a time. The neuron that wins the competition is called winner-take-all neuron.

In the simplest form of competitive learning, an ANN has a single layer of output neurons, each of which is fully connected to the input nodes. The network may include feedback connections among the neurons, as indicated in Figure 7.12. In the network architecture described herein, the feedback connections perform *lateral inhibition*, with each neuron tending to inhibit the neuron to which it is laterally connected. In contrast, the feedforward synaptic connections in the network of Figure 7.12 are all *excitatory*.

For a neuron k to be the winning neuron, its net value net_k for a specified input sample $X = \{x_1, x_2, \ldots, x_n\}$ must be the largest among all the neurons in the network. The output signal y_k of the winning neuron k is set equal to one; the outputs of all other neurons that lose the competition are set equal to 0. We thus write

$$y_k = \begin{cases} 1 & \text{if } \text{net}_k > \text{net}_j \quad \text{for all j, } j \neq k \\ 0 & \text{otherwise} \end{cases}$$

where the induced local value net_k represents the combined action of all the forward and feedback inputs to neuron k.

Let w_{kj} denote the synaptic weights connecting input node j to neuron k. A neuron then learns by shifting synaptic weights from its inactive input nodes to its active input nodes. If a particular neuron wins the competition, each input node of that neuron relinquishes some proportion of its synaptic weight, and the weight relinquished is then distributed among the active input nodes. According to the standard, competitive-learning rule, the change Δw_{kj} applied to synaptic weight w_{kj} is defined by

$$\Delta w_{kj} = \begin{cases} \eta(x_j - w_{kj}) & \text{if neuron k wins the competition} \\ 0 & \text{if neuron k loses the competition} \end{cases}$$

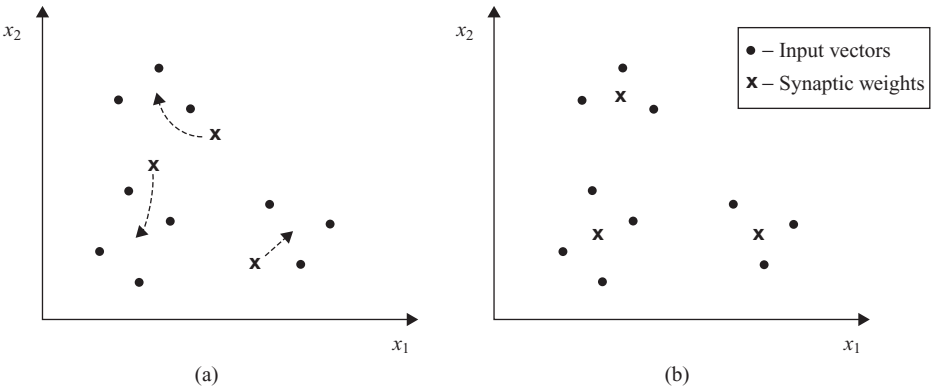


Figure 7.13. Geometric interpretation of competitive learning. (a) Initial state of the network; (b) final state of the network.

where η is the learning-rate parameter. The rule has the overall effect of moving the synaptic weights of the winning neuron toward the input pattern X. We may use the geometric analogy represented in Figure 7.13 to illustrate the essence of competitive learning.

Each output neuron discovers a cluster of input samples by moving its synaptic weights to the center of gravity of the discovered cluster. Figure 7.13 illustrates the ability of a neural network to perform clustering through competitive learning. During the competitive-learning process, similar samples are grouped by the network and represented by a single artificial neuron at the output. This grouping, based on data correlation, is done automatically. For this function to be performed in a stable way, however, the input samples must fall into sufficiently distinct groups. Otherwise, the network may be unstable.

Competitive (or winner-take-all) neural networks are often used to cluster input data where the number of output clusters is given in advance. Well-known examples of ANNs used for clustering based on unsupervised inductive learning include Kohonen's learning vector quantization (LVQ), self-organizing map (SOM), and networks based on adaptive-resonance theory models. Since the competitive network discussed in this chapter is very closely related to the Hamming networks, it is worth reviewing the key concepts associated with this general and very important class of ANNs. The Hamming network consists of two layers. The first layer is a standard, feedforward layer, and it performs a correlation between the input vector and the preprocessed output vector. The second layer performs a competition to determine which of the preprocessed output vectors is closest to the input vector. The index of the second-layer neuron with a stable, positive output (the winner of the competition) is the index of the prototype vector that best matches the input.

Competitive learning makes efficient adaptive classification, but it suffers from a few methodological problems. The first problem is that the choice of learning rate η forces a trade-off between speed of learning and the stability of the final weight factors. A learning rate near 0 results in slow learning. Once a weight vector reaches the center

of a cluster, however, it will tend to stay close to the center. In contrast, a learning rate near 1 results in fast but unstable learning. A more serious stability problem occurs when clusters are close together, which causes weight vectors also to become close, and the learning process switches its values and corresponding classes with each new example. Problems with the stability of competitive learning may occur also when a neuron's initial weight vector is located so far from any input vector that it never wins the competition, and therefore it never learns. Finally, a competitive-learning process always has as many clusters as it has output neurons. This may not be acceptable for some applications, especially when the number of clusters is not known or if it is difficult to estimate it in advance.

The following example will trace the steps in the computation and learning process of competitive networks. Suppose that there is a competitive network with three inputs and three outputs. The task is to group a set of 3-D input samples into three clusters. The network is fully connected; there are connections between all inputs and outputs and there are also lateral connections between output nodes. Only local feedback weights are equal to 0, and these connections are not represented in the final architecture of the network. Output nodes are based on a linear-activation function with the bias value for all nodes equal to zero. The weight factors for all connections are given in Figure 7.14, and we assume that the network is already trained with some previous samples.

Suppose that the new sample vector X has components

$$X = \{x_1, x_2, x_3\} = \{1, 0, 1\}$$

In the first, forward phase, the temporary outputs for competition are computed through their excitatory connections and their values are

$$\text{net}_1^* = 0.5 \cdot x_1 + (-0.5) \cdot x_3 = 0.5 \cdot 1 - 0.5 \cdot 1 = 0$$

$$\text{net}_2^* = 0.3 \cdot x_1 + 0.7 \cdot x_2 = 0.3 \cdot 1 + 0.7 \cdot 0 = 0.3$$

$$\text{net}_3^* = 0.2 \cdot x_2 + (-0.2) \cdot x_3 = 0.2 \cdot 0 - 0.2 \cdot 1 = -0.2$$

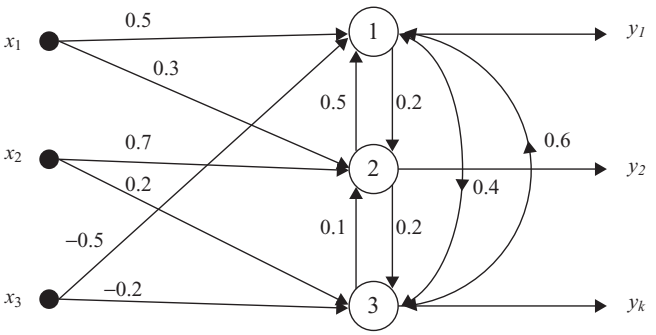


Figure 7.14. Example of a competitive neural network.

and after including lateral inhibitory connections:

$$\text{net}_1 = \text{net}_1 * + 0.5 \cdot 0.3 + 0.6 \cdot (-0.2) = 0.03$$

$$\text{net}_2 = \text{net}_2 * + 0.2 \cdot 0 + 0.1 \cdot (-0.2) = 0.28 \text{ (maximum)}$$

$$\text{net}_3 = \text{net}_3 * + 0.4 \cdot 0 + 0.2 \cdot 0.3 = -0.14$$

Competition between outputs shows that the highest output value is net_2, and it is the winner. So the final outputs from the network for a given sample will be

$$Y = \{y_1, y_2, y_3\} = \{0, 1, 0\}$$

Based on the same sample, in the second phase of competitive learning, the procedure for a weight factor's correction (only for the winning node y_2) starts. The results of the adaptation of the network, based on learning rate $\eta = 0.2$, are new weight factors:

$$w_{12} = 0.3 + 0.2\ (1 - 0.3) = 0.44$$

$$w_{22} = 0.7 + 0.2\ (0 - 0.7) = 0.56$$

$$w_{32} = 0.0 + 0.2\ (1 - 0.0) = 0.20$$

The other weight factors in the network remain unchanged because their output nodes were not the winners in the competition for this sample. New weights are the results of a competitive-learning process only for one sample. The process repeats iteratively for large training data sets.

7.7 SOMs

SOMs, often called Kohonen maps, are a data visualization technique introduced by University of Helsinki Professor Teuvo Kohonen,. The main idea of the SOMs is to project the n-dimensional input data into some representation that could be better understood visually, for example, in a 2-D image map. The SOM algorithm is not only a heuristic model used to visualize, but also to explore linear and nonlinear relationships in high-dimensional data sets. SOMs were first used in the 1980s for speech-recognition problems, but later they become a very popular and often used methodology for a variety of clustering and classification-based applications.

The problem that data visualization attempts to solve is: Humans simply cannot visualize high-dimensional data, and SOM techniques are created to help us visualize and understand the characteristics of these dimensional data. The SOM's output emphasizes on the salient features of the data, and subsequently leads to the automatic formation of clusters of similar data items. SOMs are interpreted as unsupervised neural networks (without teacher) and they are solving clustering problem by visualization of clusters. As a result of a learning process, SOM is used as an important visualization and data-reduction aid as it gives a complete picture of the data; similar data items are transformed in lower dimension but still automatically group together.

The way SOMs perform dimension reduction is by producing an output map of usually one or two dimensions that plot the similarities of the data by grouping similar

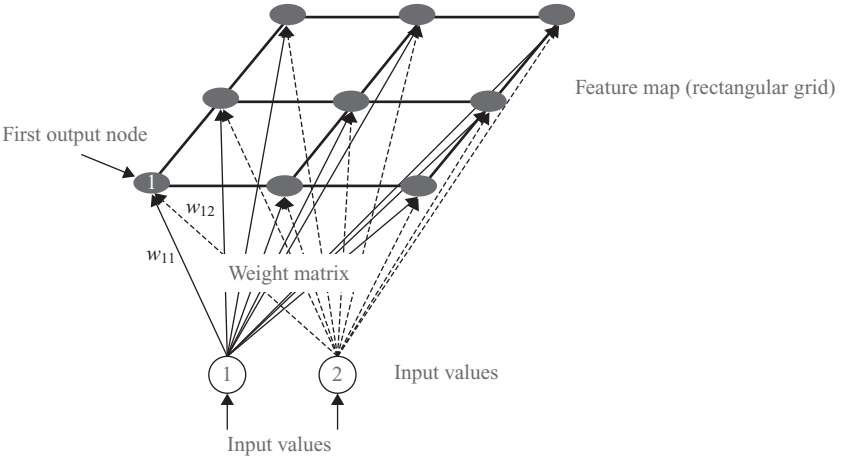


Figure 7.15. SOM with 2-D Input, 3 × 3 Output.

data items on the map. Through these transformations SOMs accomplish two goals: They reduce dimensions and display similarities. Topological relationships in input samples are preserved while complex multidimensional data can be represented in a lower dimensional space.

The basic SOM can be visualized as a neural network whose nodes become specifically tuned to various input sample patterns or classes of patterns in an orderly fashion. Nodes with weighted connections are sometimes referred to as neurons since SOMs are actually a specific type of ANNs. SOM is represented as a single layer feedforward network where the output neurons are arranged usually in a 2-D topological grid. The output grid can be either rectangular or hexagonal. In the first case each neuron except borders and corners has four nearest neighbors, in the second there are six. The hexagonal map requires more calculations, but final visualization provides a smoother result. Attached to every output neuron there is a weight vector with the same dimensionality as the input space. Each node i has a corresponding weight vector $w_i = \{w_{i1}, w_{i2}, \ldots, w_{id}\}$ where d is a dimension of the input feature space.

The structure of the SOM outputs may be a 1-D array or a 2-D matrix, but may also be more complex structures in 3-D such as cylinder or toroid. Figure 7.15 shows an example of a simple SOM architecture with all interconnections between inputs and outputs.

An important part of an SOM technique is the data. These are the samples used for SOM learning. The learning process is competitive and unsupervised, meaning that no teacher is needed to define the correct output for a given input. Competitive learning is used for training the SOM, that is, output neurons compete among themselves to share the input data samples. The winning neuron, with weights w_w, is a neuron that is the "closest" to the input example x among all other m neurons in the defined metric:

$$d(x, w_w) = \arg\min_{1 \le j \le m} d(x, w_j)$$

In the basic version of SOMs, only one output node (winner) at a time is activated corresponding to each input. The winner-take-all approach reduces the distance between winner's node weight vector and the current input sample, making the node closer to be "representative" for the sample. The SOM learning procedure is an iterative process that finds the parameters of the network (weights w) in order to organize the data into clusters that keep the topological structure. Thus, the algorithm finds an appropriate projection of high-dimensional data into a low-dimensional space.

The first step of the SOM learning is the initialization of the neurons' weights where two methods are widely used. Initial weights can be taken as the coordinates of randomly selected m points from the data set (usually normalized between 0 and 1), or small random values can be sampled evenly from the input data subspace spanned by the two largest principal component eigenvectors. The second method can increase the speed of training but may lead to a local minima and miss some nonlinear structures in the data.

The learning process is performed after initialization where training data set is submitted to the SOM one by one, sequentially, and usually in several iterations. Each output with its connections, often called a cell, is a node containing a template against which input samples are matched. All output nodes are compared with the same input sample in parallel, and SOM computes the distances between each cell and the input. All cells compete, so that only the cell with the closest match between the input and its template produces an active output. Each node therefore acts like a separate decoder or pattern detector for the same input sample, and the winning node is commonly known as the Best Matching Unit (BMU).

When the winning node is determined for a given input sample, the learning phase adapts the weight vectors in the SOM. Adaptation of the weight vectors for each output occurs through a process similar to competitive learning except that subsets of nodes are adapted at each learning step in order to produce topologically ordered maps. The "winning" node BMU aligns its own weight vector with the training input and hence becomes sensitive to it and will provide maximum response if it is shown to the network again after training. Nodes in the neighborhood set of the "winning" node must also be modified in a similar way to create regions of nodes that will respond to related samples. Nodes outside the neighborhood set remain unchanged. Figure 7.16a gives an example of 2-D matrix outputs for SOM. For the given BMU the neighborhood is defined as a 3×3 matrix of nodes surrounding BMU.

Every node within the BMU's neighborhood (including the BMU) has its weight vector adjusted according to the following equation in the iterative training process:

$$w_i(t+1) = w_i(t) + h_i(t)[x(t) - w_i(t)]$$

where $h_i(t)$ is a so-called neighborhood function. It is defined as a function of time t or more precisely a training iteration, and it specifies the neighborhood area of the ith neuron. It has been found experimentally that in order to achieve global ordering of the map the neighborhood set around the winning node should initially be large to quickly produce a rough mapping. With the increased number of iterations through the training set data, the neighborhood should be reduced to force a more localized adaptation of the network. This is done so the input samples can first move to an area of SOM where

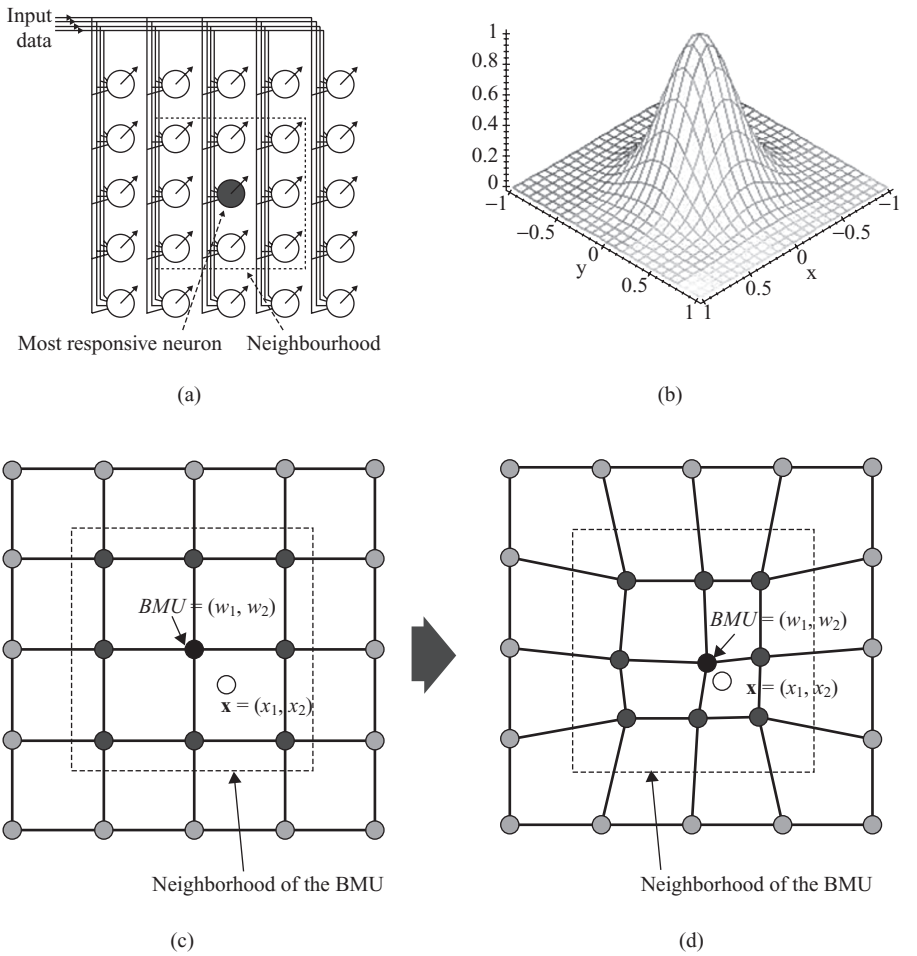


Figure 7.16. Characteristics of SOM learning process. (a) SOM: BMU and neighborhood; (b) the radius of the neighborhood diminishes with each sample and iteration; (c) BEFORE learning, rectangular grid of SOM; (d) AFTER learning, rectangular grid of SOM.

they will probably be, and then they will more precisely determine the position. This process is similar to coarse adjustment followed by fine-tuning (Fig. 7.17). The radius of the neighborhood of the BMU is therefore dynamic. To do this SOM can use, for example, the *exponential decay function* that reduces the radius dynamically with each new iteration. The graphical interpretation of the function is given in Figure 7.16b.

The simplest neighborhood function, which refers to a neighborhood set of nodes around the BMU node i, is a monotonically decreasing Gaussian function:

$$h_i(t) = a(t)\exp\left(\frac{-d(i,w)}{2\sigma^2(t)}\right)$$

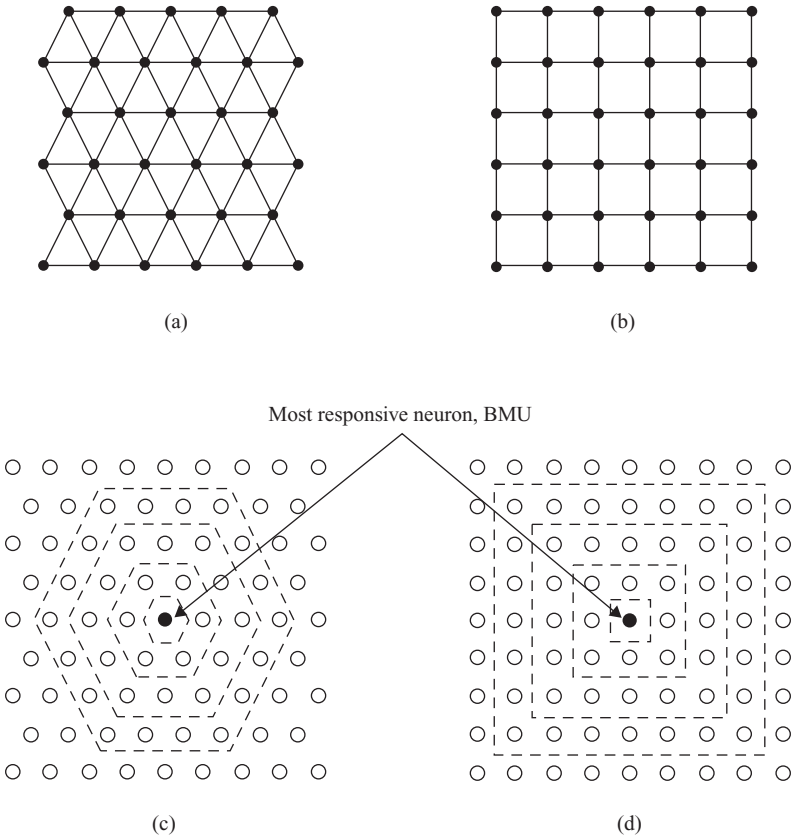


Figure 7.17. Coarse adjustment followed by fine-tuning through the neighborhood. (a) Hexagonal grid; (b) rectangular grid; (c) neighborhood in a hexagonal grid; (d) neighborhood in a rectangular grid.

where $\alpha(t)$ is a learning rate ($0 < \alpha(t) < 1$), and the width of the kernel $\sigma(t)$ is a monotonically decreasing function of time as well, and t is the current time step (iteration of the loop). While the process will adapt all weight vectors within the current neighborhood region, including those of the winning neuron, those outside this neighborhood are left unchanged. The initial radius is set high, some values near the width or height of the map. As a result, at the early stage of training when the neighborhood is broad and covers almost all the neurons, the self-organization takes place at the global scale. As the iterations continue, the base goes toward the center, so there are fewer neighbors as time progresses. At the end of training, the neighborhood shrinks to 0 and only BMU neuron updates its weights. The network will generalize through the process to organize similar vectors (which it has not previously seen) spatially close at the SOM outputs.

Apart from reducing the neighborhood, it has also been found that quicker convergence of the SOM algorithm is obtained if the adaptation rate of nodes in the network is reduced over time. Initially the adaptation rate should be high to produce coarse

clustering of nodes. Once this coarse representation has been produced, however, the adaptation rate is reduced so that smaller changes to the weight vectors are made at each node and regions of the map become fine-tuned to the input-training vectors. Therefore, every node within the BMU's neighborhood including the BMU has its weight vector adjusted through the learning process. The previous equation for weight factors correction $h_i(t)$ may include an exponential decrease of "winner's influence" introducing $\alpha(t)$ also as a monotonically decreasing function.

The number of output neurons in an SOM (i.e., map size) is important to detect the deviation of the data. If the map size is too small, it might not explain some important differences that should be detected between input samples. Conversely, if the map size is too big, the differences are too small. In practical applications, if there is no additional heuristics, the number of output neurons in an SOM can be selected using iterations with different SOM architectures.

The main advantages of SOM technology are as follows: presented results are very easy to understand and interpret; technology is very simple for implementation; and most important, it works well in many practical problems. Of course, there are also some disadvantages. SOMs are computationally expensive; they are very sensitive to measure of similarity; and finally, they are not applicable for real-world data sets with missing values. There are several possible improvements in implementations of SOMs. To reduce the number of iterations in a learning process, good initialization of weight factors is essential. *Principal components of input data* can make computation of the SOM orders of magnitude faster. Also, practical experience shows that hexagonal grids give output results with a better quality. Finally, selection of distance measure is important as in any clustering algorithm. Euclidean distance is almost standard, but that does not mean that it is always the best. For an improved quality (isotropy) of the display, it is advisable to select the grid of the SOM units as *hexagonal*.

The SOMs have been used in large spectrum of applications such as automatic speech recognition, clinical data analysis, monitoring of the condition of industrial plants and processes, classification from satellite images, analysis of genetic information, analysis of electrical signals from the brain, and retrieval from large document collections. Illustrative examples are given in Figure 7.18.

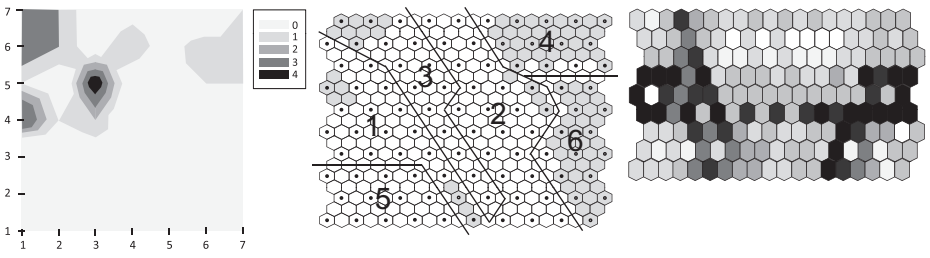


Figure 7.18. SOM applications. (a) Drugs binding to human cytochrome; (b) interest rate classification; (c) analysis of book-buying behavior.

7.8 REVIEW QUESTIONS AND PROBLEMS

1. Explain the fundamental differences between the design of an ANN and "classical" information-processing systems.

2. Why is fault-tolerance property one of the most important characteristics and capabilities of ANNs?

3. What are the basic components of the neuron's model?

4. Why are continuous functions such as log-sigmoid or hyperbolic tangent considered common activation functions in real-world applications of ANNs?

5. Discuss the differences between feedforward and recurrent neural networks.

6. Given a two-input neuron with the following parameters: bias b = 1.2, weight factors W = [w1, w2] = [3, 2], and input vector X = $[-5, 6]^T$; calculate the neuron's output for the following activation functions:

(a) a symmetrical hard limit
(b) a log-sigmoid
(c) a hyperbolic tangent

7. Consider a two-input neuron with the following weight factors W and input vector X:

$$W = [3, 2] \quad X = [-5, 7]^T$$

We would like to have an output of 0.5.

(a) Is there a transfer function from Table 9.1 that will do the job if the bias is zero?
(b) Is there a bias that will do the job if the linear-transfer function is used?
(c) What is the bias that will do the job with a log-sigmoid–activation function?

8. Consider a classification problem defined with the set of 3-D samples X, where two dimensions are inputs and the third one is the output.

X:	I_1	I_2	O
	−1	1	1
	0	0	1
	1	−1	1
	1	0	0
	0	1	0

(a) Draw a graph of the data points X labeled according to their classes. Is the problem of classification solvable with a single-neuron perceptron? Explain the answer.
(b) Draw a diagram of the perceptron you would use to solve the problem. Define the initial values for all network parameters.
(c) Apply single iteration of the delta-learning algorithm. What is the final vector of weight factors?

9. The one-neuron network is trained to classify input–output samples:

1	0	1
1	1	−1
0	1	1

Show that this problem cannot be solved unless the network uses a bias.

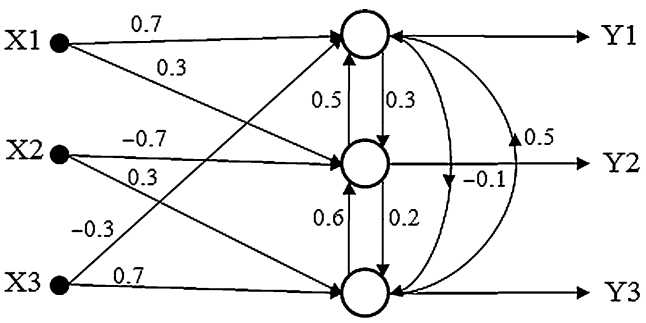
10. Consider the classification problem based on the set of samples X:

X:	I_1	I_2	O
	−1	1	1
	−1	−1	1
	0	0	0
	1	0	0

(a) Draw a graph of the data points labeled according to their classification. Is the problem solvable with one artificial neuron? If yes, graph the decision boundaries.
(b) Design a single-neuron perceptron to solve this problem. Determine the final weight factors as a weight vector orthogonal to the decision boundary.
(c) Test your solution with all four samples.
(d) Using your network classify the following samples: (–2, 0), (1, 1), (0, 1), and (–1, –2).
(e) Which of the samples in (d) will always be classified the same way, and for which samples classification may vary depending on the solution?

11. Implement the program that performs the computation (and learning) of a single-layer perceptron.

12. For the given competitive network:



(a) find the output vector [Y1, Y2, Y3] if the input sample is [X1, X2, X3] = [1, −1, −1];
(b) what are the new weight factors in the network?

13. Search the Web to find the basic characteristics of publicly available or commercial software tools that are based on ANNs. Document the results of your search. Which of them are for learning with a teacher, and which are support learning without a teacher?

14. For a neural network, which one of these structural assumptions is the one that most affects the trade-off between underfitting (i.e., a high-bias model) and overfitting (i.e., a high-variance model):

(a) the number of hidden nodes,
(b) the learning rate,
(c) the initial choice of weights, or
(d) the use of a constant-term unit input.

15. Is it true that the Vapnik-Chervonenkis (VC) dimension of a perceptron is smaller than the VC dimension of a simple linear support vector machine (SVM)? Discuss your answer.

7.9 REFERENCES FOR FURTHER STUDY

Engel, A., C. Van den Broeck, *Statistical Mechanics of Learning*, Cambridge University Press, Cambridge, UK, 2001.

The subject of this book is the contribution made to machine learning over the last decade by researchers applying the techniques of statistical mechanics. The authors provide a coherent account of various important concepts and techniques that are currently only found scattered in papers. They include many examples and exercises to make a book that can be used with courses, or for self-teaching, or as a handy reference.

Haykin, S., *Neural Networks and Learning Machines*, 3rd edition, Prentice Hall, Upper Saddle River, NJ, 2009.

Fluid and authoritative, this well-organized book represents the first comprehensive treatment of neural networks from an engineering perspective, providing extensive, state-of-the-art coverage that will expose readers to the myriad facets of neural networks and help them appreciate the technology's origin, capabilities, and potential applications. The book examines all the important aspects of this emerging technology, covering the learning process, backpropagation, radial basis functions, recurrent networks, self-organizing systems, modular networks, temporal processing, neurodynamics, and VLSI implementation. It integrates computer experiments throughout to demonstrate how neural networks are designed and perform in practice. Chapter objectives, problems, worked examples, a bibliography, photographs, illustrations, and a thorough glossary all reinforce concepts throughout. New chapters delve into such areas as SVMs, and reinforcement learning/neurodynamic programming, plus readers will find an entire chapter of case studies to illustrate the real-life, practical applications of neural networks. A highly detailed bibliography is included for easy reference. It is the book for professional engineers and research scientists.

Heaton, J., *Introduction to Neural Networks with Java*, Heaton Research, St. Louis, MO, 2005.

Introduction to *Neural Networks with Java* introduces the Java programmer to the world of neural networks and artificial intelligence (AI). Neural-network architectures such as the feedforward backpropagation, Hopfield, and Kohonen networks are discussed. Additional AI

topics, such as Genetic Algorithms and Simulated Annealing, are also introduced. Practical examples are given for each neural network. Examples include the Traveling Salesman problem, handwriting recognition, fuzzy logic, and learning mathematical functions. All Java source code can be downloaded online. In addition to showing the programmer how to construct these neural networks, the book discusses the Java Object Oriented Neural Engine (JOONE). JOONE is a free open source Java neural engine.

Principe, J. C., R. Mikkulainen, *Advances in Self-Organizing Maps, Series: Lecture Notes in Computer Science*, Vol. 5629, Springer, New York, 2009.

This book constitutes the refereed proceedings of the 7th International Workshop on Advances in Self-Organizing Maps, WSOM 2009, held in St. Augustine, Florida, in June 2009. The 41 revised full papers presented were carefully reviewed and selected from numerous submissions. The papers deal with topics in the use of SOM in many areas of social sciences, economics, computational biology, engineering, time-series analysis, data visualization, and theoretical computer science.

Zurada, J. M., *Introduction to Artificial Neural Systems*, West Publishing Co., St. Paul, MN, 1992.

The book is one of the traditional textbooks on ANNs. The text grew out of a teaching effort in artificial neural systems offered for both electrical engineering and computer science majors. The author emphasizes that the practical significance of neural computation becomes apparent for large or very large-scale problems.