Chapter

5 LEARNING WITH NEURAL NETWORKS (NN)

5.1 TOWARDS COGNITIVE MACHINE

Human intelligence possesses robust attributes with complex sensory, control, affective (emotional processes), and cognitive (thought processes) aspects of information processing and decision making. There are over a hundred billion biological neurons in our central nervous system (CNS), playing a key role in these functions. CNS obtains information from the external environment via numerous natural sensory mechanisms—vision, hearing, touch, taste, and smell. With the help of cognitive computing, it assimilates the information and offers the right interpretation. The cognitive process then progresses towards some attributes, such as learning, recollection, and reasoning, which results in proper actions via muscular control.

The progress in technology based on information in recent times, has widened the capabilities and applications of computers. If we wish a machine (computer) to exhibit certain cognitive functions, such as learning, remembering, reasoning and perceiving, that humans are known to exhibit, we need to define 'information' in a general manner and develop new mathematical techniques and hardware with the ability to handle the simulation and processing of cognitive information. Mathematics, in its present form, was developed to comprehend physical processes, but cognition, as a process, does not essentially follow these mathematical laws. So what exactly is *cognitive mathematics* then? The question is rather difficult. However, scientists have converged to the understanding that if certain 'mathematical aspects' of our process of thinking are re-examined along with the 'hardware aspects' of 'the neurons'—which is the primary component of the brain—we may, to a certain level, be able to successfully emulate the process.

Biological neuronal procedures are rather complex [76], and the advancement made in terms of understanding the field with the help of experiments is raw and inadequate. However, with the help of this limited understanding of the biological processes, it has been possible to emulate some human learning behaviors, via the fields of mathematics and systems science. Neuronal information processing involves a range of complex mathematical processes and mapping functions. And they serve as a parallel-cascade computing structure in synergism. The aim of system scientists is to

create an intelligent cognitive system on the basis of this limited understanding of the brain—a system that can help human beings to perform all kinds of tasks requiring decision making. Various new computing theories of the *neural networks* field have been developing, which it is hoped will be capable of providing a *thinking machine*. Given that they are based on neural networks architecture, they should hopefully be able to create a low-level cognitive machine, which scientists have been trying to build for so long.

The subject of cognitive machines is in an exciting state of research and we believe that we are slowly progressing towards the development of truly cognitive machines.

5.1.1 From Perceptrons to deep Networks

Historically, research in neural networks was inspired by the desire to produce artificial systems capable of sophisticated 'intelligent' processing similar to the human brain. The *perceptron* is the earliest of the artificial neural networks paradigms. Frank Rosenblatt built this learning machine device in hardware in 1958. In 1959, Bernard Widrow and Marcian Hoff developed a learning rule, sometimes known as *Widrow-Haff rule*, for ADALINE (ADAptive LINear Elements). Their learning rule was simple and yet elegant.

Affected by the predominantly rosy outlook of the time, some people exaggerated the potential of neural networks. Biological comparisons were blown out of proportion. In 1969, significant limitations of perceptrons, a fundamental block for more powerful models, were highlighted by Marvin Minsky. It brought to a halt much of the activity in neural network research.

Nevertheless, a few dedicated scientists, such as Teuvo Kohonen and Stephen Grossberg, continued their efforts. In 1982, John Hopfield introduced a *recurrent*-type neural network that was based on the interaction of neurons through a feedback mechanism. The *back-propagation learning rule* arrived on the neural-network scene at approximately the same time from several independent sources (Werbos; Parker; Rumelhart, Hinton and Williams). Essentially a refinement of the Widrow-Hoff learning rule, the back-propagation learning rule provided a systematic means for training *multilayer feedforward* networks, thereby overcoming the limitations presented by Minsky. Research in the 1980s triggered a boom in the scientific community. New and better models have been proposed. A number of today's technological problems are in the areas where neural-network technology has demonstrated potential.

As the research in neural networks is evolving, more and more types of networks are being introduced. For reasonably complex problems, neural networks with back-propagation learning have serious limitations. The learning speed of these feedforward neural networks is, in general, far slower than required and it has been a major bottleneck in their applications. Two reasons behind this limitation may be: (i) the slow gradient-based learning algorithms extensively used to train neural networks, and (ii) all the parameters of the network are tuned iteratively by using learning algorithms. A new learning algorithm was proposed in 2006 by Huang, Guang-Bin et. al. [77], called *Extreme Learning Machine* (ELM), for single hidden layer feedforward neural networks, in which the weights connecting input to hidden nodes are randomly chosen and never updated and weights connecting hidden nodes to output are analytically determined. Experimental results based on real-world benchmarking function approximation (regression) and classification problems show that ELM can produce best generalization in some cases and can be thousands of times faster than traditional popular learning algorithms for feedforward neural networks.

Novel research investigations in ELM and related areas have produced a suite of machine learning techniques for (single and multi-) hidden layer feedforward networks in which hidden neurons need not be tuned. ELM theories argue that random hidden neurons capture the essence of some brain learning mechanisms. ELM has a great potential as a viable alternative technique for large-scale computing and AI (*Artificial Intelligence*).

The 'traditional' neural networks are based on what we might interpret as 'shallow' learning; in fact, this learning methodology has very little resemblance to the brain, and one might argue that it would be more fair to regard them simply as a discipline under statistics.

The subject of cognitive machines is in an exciting state of research and we believe that we are slowly progressing towards the development of truly intelligent systems. A step towards realizing strong AI has been taken through the recent research in 'deep learning'. Considering the far-reaching applications of AI, coupled with the awareness that deep learning is evolving as one of its most powerful methods, today it is not possible for one to enter the machine learning community without any knowledge of deep networks.

Deep learning algorithms are in sharp contrast to shallow learning algorithms in terms of the number of parameterized transformations a signal comes across as it spreads from input layer to the output layer. A parameterized transformation refers to a processing unit containing trainable parameters—weights and thresholds. A chain of transformations from input to output is a *Credit Assignment Path* (CAP), which describes potentially causal connections from input to output and may have varied lengths. In case of a feedforward neural network, the depth of the CAPs, and therefore, the *depth of the network*, is a number of hidden layers plus one (the output layer is also parameterized).

Today, deep learning, based on *learning representations of data*, is a significant member of family of machine learning techniques. It is possible to represent an observation, for instance an image, in various ways, such as a vector of intensity values per pixel or in a more abstract way as a set of edges, regions of particular shape, and so on. Some representations make learning tasks simpler. Deep learning aims to replace hand-crafted *features* with efficient algorithms for *supervised* or *unsupervised feature learning*, and hierarchical *feature extraction*.

The field of deep learning has been characterized in several ways. These definitions have in common:

(i) multiple layers of nonlinear processing units
(ii) the supervised/unsupervised learning of feature representations in each layer, with the layers giving rise to a hierarchy from low-level to high-level characteristics.

What a layer of nonlinear processing unit, employed in a deep learning algorithm, consists of is dependent on the problem that needs to be solved.

Deep learning is linked closely to a category of brain-development theories published by cognitive neuro scientists in the early 1990s. Some of the deep-learning representations are inspired by progresss in neuroscience and are roughly based on interpretation of information processing and communication patterns in a nervous system, such as neural coding which tries to describe the relationship between a range of stimuli and related neuronal responses in the brain.

The term 'deep learning' gained traction in the mid 2000s, after a publication by Geoffrey Hinton and Ruslan Salakhutdinov. They showed how many-layered feedforward network could

be effectively pre-trained one layer at a time. Since its resurgence, deep learning has become part of the many-of-the-art systems in various disciplines, particularly computer vision and speech recognition. The real impact of deep learning in industry began in large-scale speech recognition around 2010. Recent useful references on the subject are [78–82].

Our focus in this chapter will be on traditional neural networks [83, 84]. These networks are being used today for many real-world regression and classification problems. Also, a sound understanding of these networks is a prerequisite to learn ELM and deep learning algorithms. The two recent research developments are outside the scope of this book.

The terms 'Neural Networks (NN)' and 'Artificial Neural Networks (ANN)' are both commonly used in the literature for the same field of study. We will use the term 'Neural Networks' in this book.

Broadly speaking, AI (*Artificial Intelligence*) is any computer program that does something smart [2, 5]. *Machine learning* is a subfield of AI. That is, all machine learning counts as AI, but not all AI counts as machine learning. For example, rule-based expert systems, frame-based expert systems, knowledge graphs, evolutionary algorithms could be described by AI but none of them is in machine learning. *Deep learning* may be considered as subfield of machine learning. Deep neural networks are a set of algorithms setting new records in accuracy for many important problems. *Deep* is a technical term; it refers to number of layers in a neural network. Multiple hidden layers allow deep neural networks to learn features of the data in a hierarchy.

Deep learning may share elements of traditional machine learning, but some researchers feel that it will emerge as a *class* by itself, as a subfield of AI.

5.2 NEURON MODELS

A discussion of anthropomorphism to introduce neural network technology may be worthwhile—as it helps explain the terminology of neural networks. However, anthropomorphism can lead to misunderstanding when the metaphor is carried too far. We give here a brief description of how the brain works; a lot of details of the complex electrical and chemical processes that go on in the brain, have been ignored. A pragmatic justification for such a simplification is that by starting with a simple model of the brain, scientists have been able to achieve very useful results.

5.2.1 Biological Neuron

To the extent a human brain is understood today, it seems to operate as follows: bundles of neurons, or nerve fibers, form nerve structures. There are many different types of neurons in the nerve structure, each having a particular shape, size and length depending upon its function and utility in the nervous system. While each type of neuron has its own unique features needed for specific purposes, all neurons have two important structural components in common. These may be seen in the typical biological neuron shown in Fig. 5.1. At one end of the neuron are a multitude of tiny, filament-like appendages called *dendrites*, which come together to form larger branches and trunks where they attach to *soma*, the body of the nerve cell. At the other end of the neuron is a single filament leading out of the soma, called an *axon*, which has extensive branching on its far end. These two structures have special electrophysiological properties which are basic to the function of neurons as *information processors*, as we shall see next.

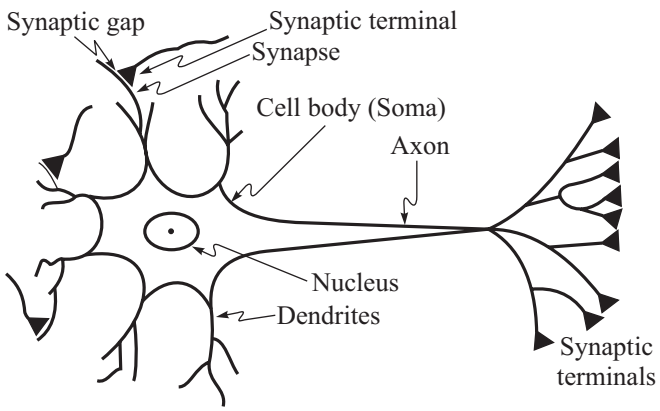


Figure 5.1 A typical biological neuron

Neurons are connected to each other via their axons and dendrites. Signals are sent through the axon of one neuron to the dendrites of other neurons. Hence, dendrites may be represented as the inputs to the neuron, and the axon as its output. Note that each neuron has many inputs through its multiple dendrites, whereas it has only one output through its single axon. The axon of each neuron forms connections with the dendrites of many other neurons, with each branch of the axon meeting exactly one dendrite of another cell at what is called a *synapse*. Actually, the axon terminals do not quite touch the dendrites of the other neurons, but are separated by a very small distance of between 50 and 200 angstroms. This separation is called the *synaptic gap*.

A conventional computer is typically a single processor acting on explicitly programmed instructions. Programmers break tasks into tiny components, to be performed in sequence rapidly. On the other hand, the brain is composed of ten billion or so neurons. Each nerve cell can interact directly with up to 200,000 other neurons (though 1000 to 10,000 is typical). In place of explicit rules that are used by a conventional computer, it is the pattern of connections between the neurons in the human brain, that seems to embody the 'knowledge' required for carrying out various information-processing tasks. In human brain, there is no equivalent of a CPU that is in overall control of the actions of all the neurons.

The brain is organized into different regions, each responsible for different functions. The largest parts of the brain are the *cerebral hemispheres*, which occupy most of the interior of the skull. They are layered structures; the most complex being the outer layer, known as the *cerebral cortex*, where the nerve cells are extremely densely packed to allow greater interconnectivity. Interaction with the environment is through the visual, auditory and motion control (muscles and glands) parts of the cortex.

In essence, neurons are tiny electrophysiological information-processing units which communicate with each other through electrical signals. The synaptic activity produces a voltage pulse on the dendrite which is then conducted into the soma. Each dendrite may have many synapses acting on it, allowing massive interconnectivity to be achieved. In the soma, the *dendrite potentials* are added. Note that neurons are able to perform more complex functions than simple addition on the inputs they receive, but considering a simple summation is a reasonable approximation.

When the *soma potential* rises above a critical threshold, the axon will fire an electrical signal. This sudden burst of electrical energy along the axon is called *axon potential* and has the form of an electrical impulse or spike that lasts about 1 msec. The magnitude of the axon potential is constant and is not related to the electrical stimulus (soma potential). However, neurons typically respond to a stimulus by firing not just one but a barrage of successive axon potentials. What varies is the frequency of axonal activity. Neurons can fire between 0 to 1500 times per second. Thus, information is encoded in the nerve signals as the *instantaneous frequency* of axon potentials and the *mean frequency* of the signal.

A synapse pairs the axon with another cell's dendrite. It discharges chemicals known as *neurotransmitters*, when its potential is increased enough by the axon potential. The triggering of the synapse may require the arrival of more than one spike. The neurotransmitters emitted by the synapse diffuse across the gap, chemically activating gates on the dendrites, which, on opening, permit the flow of charged ions. This flow of ions, changes the dendritic potential and generates voltage pulse on the dendrite, which is then conducted into the neuron body. At the synaptic junction, the number of gates that open on the dendrite is dependent on the number of neurotransmitters emitted. It seems that some synapses *excite* the dendrites they impact, while others act in a way that *inhibits* them. This results in changing the local potential of the dendrite in a positive or negative direction.

Synaptic junctions alter the effectiveness with which the signal is transmitted; some synapses are good junctions and pass a large signal across, whilst others are very poor, and allow very little through.

Essentially, each neuron receives signals from a large number of other neurons. These are the inputs to the neuron which are 'weighted'. That is, some signals are stronger than others. Some signals excite (are positive), and others inhibit (are negative). The effects of all weighted inputs are summed. If the sum is equal to or greater than the *threshold* for the neuron, the neuron *fires* (gives output). This is an 'all-or-nothing' situation. Because the neuron either fires or does not fire, the *rate of firing*, not the amplitude, conveys the magnitude of information.

The ease of transmission of signals is altered by activity in the nervous system. The neural pathway between two neurons is susceptible to fatigue, oxygen deficiency, and agents like anesthetics. These events create a resistance to the passage of impulses. Other events may increase the rate of firing. This ability to adjust signals is a mechanism for *learning*.

After carrying a pulse, an axon fiber is in a condition of complete non-excitability for a specific time period known as the *refractory period*. During this interval, the nerve conducts no signals, irrespective of how intense the excitation is. Therefore, we could segregate the time scale into successive intervals, each equal to the length of the refractory period. This will permit a discrete-time description of the neurons' performance in terms of their states at discrete-time instances.

5.2.2 Artificial Neuron

Artificial neurons bear only a modest resemblance to real things. They model approximately three of the processes that biological neurons perform (there are at least 150 processes performed by neurons in the human brain).

An artificial neuron

(i) evaluates the input signals, determining the strength of each one;
(ii) calculates a total for the combined input signals and compares that total to some threshold level; and
(iii) determines what the output should be.

Input and Outputs

Just as there are many inputs (stimulation levels) to a biological neuron, there should be many input signals to our artificial neuron (AN). All of them should come to our AN simultaneously. In response, a biological neuron either 'fires' or 'doesn't fire' depending upon some *threshold* level. Our AN will be allowed a single output signal, just as is present in a biological neuron: many inputs, one output (Fig. 5.2).

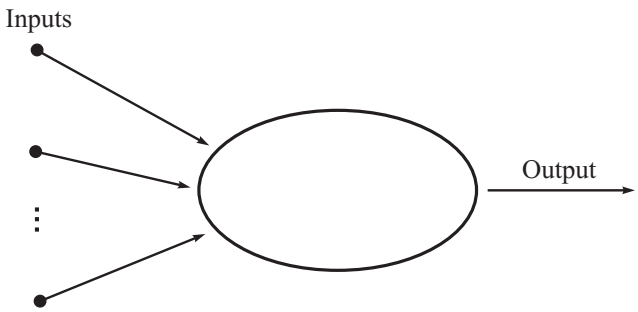


Figure 5.2 Many inputs, one output model of a neuron

Weighting Factors

Each input will be given a relative weighting, which will affect the impact of that input (Fig. 5.3). This is something like varying synaptic strengths of the biological neurons—some inputs are more important than others in the way they combine to produce an impulse. Weights are adaptive coefficients within the network, that determine the intensity of the input signal. In fact, this adaptability of connection strength is precisely what provides neural networks their ability to learn and store information, and, consequently, is an essential element of all neuron models.

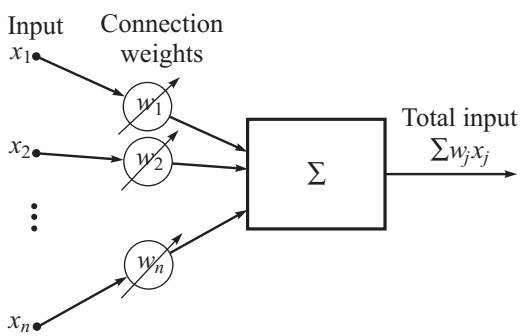


Figure 5.3 A neuron with weighted inputs

Excitatory and inhibitory inputs are represented simply by positive or negative connection weights, respectively. Positive inputs promote the firing of the neuron, while negative inputs tend to keep the neuron from firing.

Mathematically, we could look at the inputs and the weights on the inputs as vectors.

The *input vector*

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{5.1a}$$

and the connection *weight vector*

$$\mathbf{w}^T = [w_1 \; w_2 \; \ldots \; w_n] \tag{5.1b}$$

The total input signal is the product of these vectors. The result is a scalar

$$\sum_{j=1}^{N} w_j \, x_j = \mathbf{w}^T \mathbf{x} \tag{5.1c}$$

Activation Functions

Although most neuron models sum their input signals in basically the same manner, as described above, they are not all identical in terms of how they produce an output response from this input. Artificial neurons use an *activation function*, often called a *transfer function*, to compute their activation as a function of total input stimulus. Several different functions may be used as activation functions, and, in fact, the most distinguishing feature between existing neuron models is precisely which function they employ.

We will, shortly, take a closer look at the activation functions. We first build a neuron model, assuming that the transfer function has a threshold behavior, which is, in fact, the type of response exhibited by biological neurons: when the total stimulus exceeds a certain threshold value θ, a constant output is produced, while no output is generated for input levels below the threshold. Figure 5.4a shows this neuron model. In this diagram, the neuron has been represented in such a way that the correspondence of each element with its biological counterpart may be easily seen.

Equivalently, the threshold value can be subtracted from the weighted sum and the resulting value compared to zero; if the result is positive, then output a 1, else output a 0. This is shown in Fig. 5.4b; note that the shape of the function is the same but now the jump occurs at zero. The threshold effectively adds an *offset* to the weighted sum.

An alternative way of achieving the same effect is to take the threshold out of the body of the model neuron, and connect it to an extra input value that is fixed to be 'on' all the time. In this case, rather than subtracting the threshold value from the weighted sum, the extra input of +1 is multiplied by a weight and added in a manner similar to other inputs—this is known as *biasing* the neuron. Figure 5.4c shows a neuron model with a bias term. Note that we have taken constant input '1' with an adaptive weight 'w_0' in our model.

The first formal definition of a synthetic neuron model, based on the highly simplified considerations of the biological neuron, was formulated by McCulloch and Pitts (1943). The two-port model (inputs—activation value—output mapping) of Fig. 5.4 is essentially the *MP neuron model*. It is important to look at the features of this unit—which is an important and popular neural network building block.

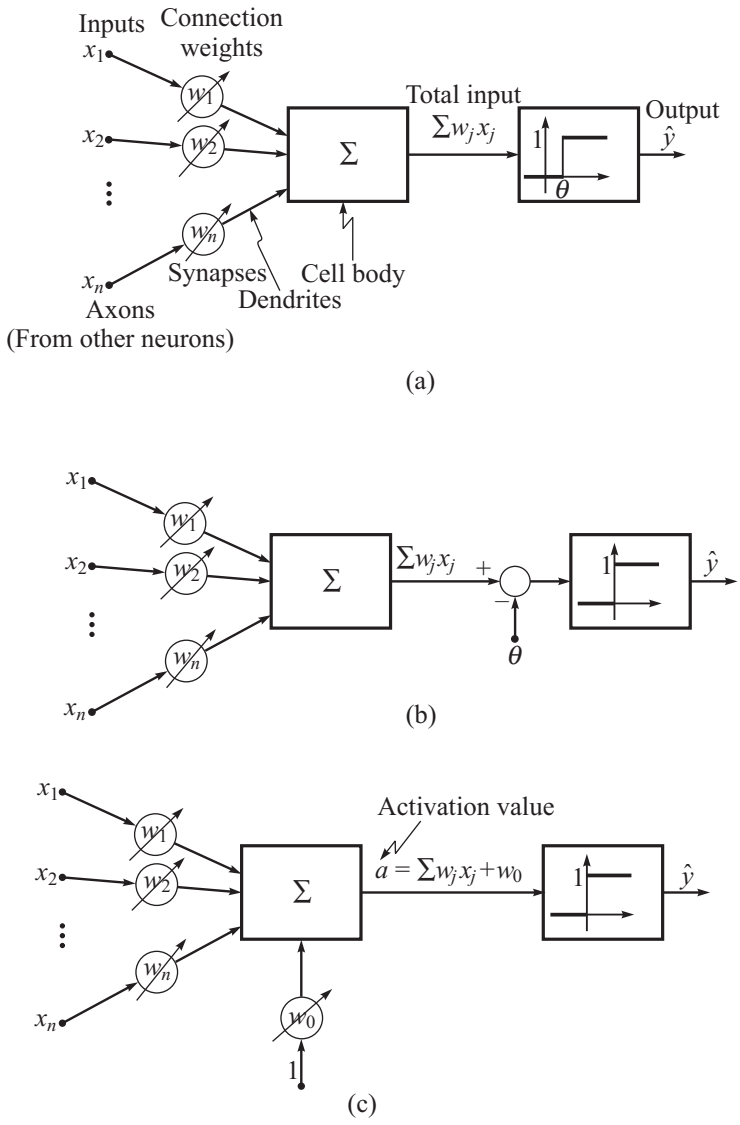


Figure 5.4 The MP neuron model

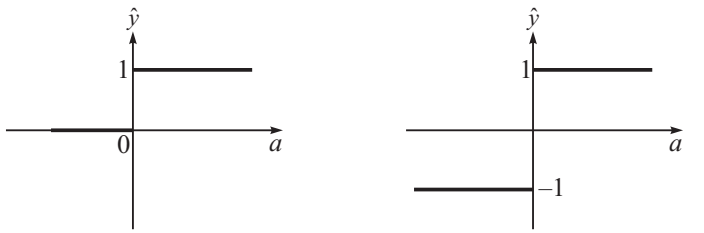
It is a basic unit, thresholding a weighted sum of its inputs to get an output. It does not particularly consider the complex patterns and timings of the real nervous activity in real neural systems. It

does not have any of the complex characteristics existing in the body of biological neurons. It is, therefore, a *model*, and not a *copy* of a real neuron.

The MP artificial neuron model involves two important processes:

(i) Forming net activation by combining inputs. The input values are amalgamated by a weighted additive process to achieve the neuron activation value a (refer to Fig. 5.4c).
(ii) Mapping this activation value a into the neuron output $\hat{y}$. This mapping from activation to output may be characterized by an 'activation' or 'squashing' function.

For the activation functions that implement input-to-output compression or squashing, the range of the function is less than that of the domain. There is some physical basis for this desirable characteristic. Recall that in a biological neuron, there is a limited range of output (spiking frequencies). In the MP model, where DC levels replace frequencies, the squashing function serves to limit the output range. The squashing function shown in Fig. 5.5a limits the output values to {0, 1}, while that in Fig. 5.5b limits the output values to {–1, 1}. The activation function of Fig. 5.5a is called *unipolar*, while that in Fig. 5.5b is called *bipolar* (both positive and negative responses of neurons are produced).



(a) Unipolar squashing function (b) Bipolar squashing function

Figure 5.5

5.2.3 Mathmatical Model

From the earlier discussion, it is evident that the artificial neuron is really nothing more than a simple mathematical equation for calculating an output value from a set of input values. From now onwards, we will be more on a mathematical footing; the reference to biological similarities will be reduced. Therefore, names like a *processing element*, a *unit*, a *node*, a *cell*, etc., may be used for the neuron. A neuron model (a processing element/a unit/a node/a cell of our neural network), will be represented as follows:

The input vector

$$\mathbf{x} = [x_1\ x_2\ \ldots\ x_n]^T;$$

the connection weight vector

$$\mathbf{w}^T = [w_1\ w_2\ \ldots\ w_n];$$

the unity-input weight w_0 (bias term), and the output $\hat{y}$ of the neuron are related by the following equation:

$$\hat{y} = \sigma(\mathbf{w}^T \mathbf{x} + w_0) = \sigma\left(\sum_{j=1}^{n} w_j x_j + w_0 \right) \tag{5.2}$$

where $\sigma(\cdot)$ is the activation function (transfer function) of the neuron.

The weights are always adaptive. We can simplify our diagram as in Fig. 5.6a; adaptation need not be specifically shown in the diagram.

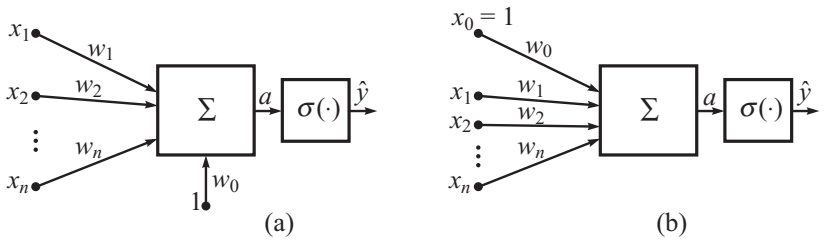


Figure 5.6 Mathematical model of a neuron (perceptron)

The bias term may be absorbed in the input vector itself as shown in Fig. 5.6b.

$$\hat{y} = \sigma(a)$$

$$= \sigma\left(\sum_{j=0}^{n} w_j x_j \right); x_0 = 1 \tag{5.3a}$$

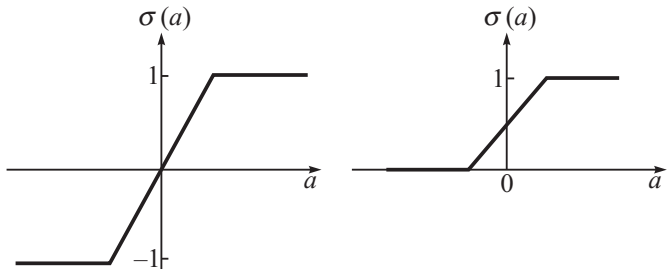
$$= \sigma\left(\sum_{j=1}^{n} w_j x_j + w_0 \right) = \sigma(\mathbf{w}^T \mathbf{x} + w_0) \tag{5.3b}$$

In the literature, this model of an artificial neuron is also referred to as a *perceptron* (the name was given by Rosenblatt in 1958).

The expressions for the neuron output $\hat{y}$ are referred to as the *cell recall mechanism*. They describe how the output is reconstructed from the input signals and the values of the cell parameters.

The artificial neural systems under investigation and experimentation today, employ a variety of activation functions that have more diversified features than the one presented in Fig. 5.5. Below, we introduce the main activation functions that will be used later in this chapter.

The MP neuron model shown in Fig. 5.4 used the *hard-limiting activation function*. When artificial neurons are cascaded together in layers (discussed in the next section), it is more common to use a *soft-limiting activation function*. Figure 5.7a shows a possible bipolar soft-limiting semilinear activation function. This function is, more or less, the ON-OFF type, as before, but has a sloping region in the middle. With this smooth thresholding function, the value of the output will be practically 1 if the weighted sum exceeds the threshold by a huge margin and, conversely, it will be practically –1 if the weighted sum is much less than the threshold value. However, if the threshold and the weighted sum are almost the same, the output from the neuron will have a value somewhere between the two extremes. This means that the output from the neuron can be related to its inputs in a more useful and informative way. Figure 5.7b shows a unipolar soft-limiting function.



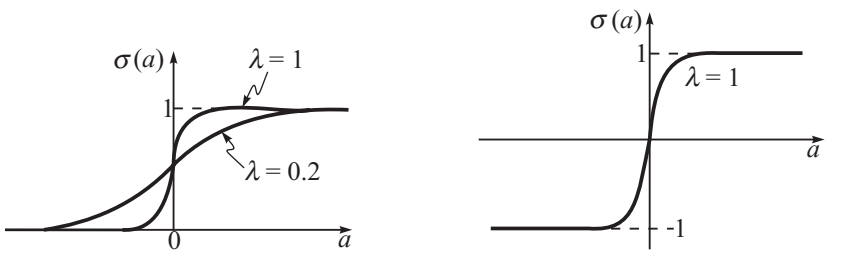
(a) Bipolar (b) Unipolar

Figure 5.7 Soft-limiting activation functions.

For many training algorithms (discussed in later sections), the derivative of the activation function is needed; therefore, the activation function selected must be differentiable. The *logistic* or *sigmoid* function, which satisfies this requirement, is the most commonly used soft-limiting activation function. The sigmoid function (Fig. 5.8a):

$$\sigma(a) = \frac{1}{1 + e^{-\lambda a}} \tag{5.4}$$

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



(a) Sigmoid functions (b) Hyperbolic tangent function

Figure 5.8

The sigmoid function is unipolar. A bipolar function with similar characteristics is a *hyperbolic tangent* (Fig. 5.8b):

$$\sigma(a) = \frac{1 - e^{-\lambda a}}{1 + e^{-\lambda a}} = \tanh\left(\tfrac{1}{2}\,\lambda a\right) \tag{5.5}$$

The biological basis of these activation functions can easily be established. It is known that neurons located in different parts of the nervous system have different characteristics. The neurons of the ocular motor system have a sigmoid characteristic, while those located in the visual area

have a Gaussian characteristic. As we said earlier, anthropomorphism can lead to misunderstanding when the metaphor is carried too far. It is now a well-known result in neural network theory that a two-layer neural network is capable of solving any classification problem. It has also been shown that a two-layer network is capable of solving any nonlinear function approximation problem [3, 83]. This result does not require the use of sigmoid nonlinearity. The proof assumes only that nonlinearity is a continuous, smooth, monotonically increasing function that is bounded above and below. Thus, numerous alternatives to sigmoid could be used, without a biological justification. In addition, the above result does not require that the nonlinearity be present in the second (output) layer. It is quite common to use linear output nodes since this tends to make learning easier. In other words,

$$\sigma(a) = \lambda a;\ \lambda > 0 \tag{5.6}$$

is used as an activation function in the output layer. Note that this function does not 'squash' (compress) the range of output.

Our focus in this chapter will be on two-layer *perceptron networks* with the first (hidden) layer having *log-sigmoid*

$$\sigma(a) = \frac{1}{1+e^{-a}} \tag{5.7a}$$

or *tan-sigmoid*

$$\sigma(a) = \frac{1-e^{-a}}{1+e^{-a}} \tag{5.7b}$$

activation function, and the second (output) layer having *linear* activation function

$$\sigma(a) = a \tag{5.8}$$

The log-sigmoid function has historically been a very popular choice, but since it is related to the tan-sigmoid by the simple transformation

$$\sigma_{\text{log-sigmoid}} = (\sigma_{\text{tan-sigmoid}} + 1)/2 \tag{5.9}$$

both of these functions are in use in neural network models.

We have so far described two classical neuron models:

- perceptron—a neuron with sigmoidal activation function (sigmoidal function is a softer version of the original perceptron's hard limiting or threshold activation function); and
- linear neuron—a neuron with linear activation function.

5.3 NETWORK ARCHITECTURES

In the biological brain, a huge number of neurons are interconnected to form the network and perform advanced intelligent activities. The artificial neural network is built by neuron models. Many different types of artificial neural networks have been proposed, just as there are many theories on how biological neural processing works. We may classify the organization of the neural networks largely into two types: a feedforward net and a recurrent net. The feedforward net has a hierarchical structure that consists of several layers, without interconnection between neurons in each layer, and signals flow from input to output layer in one direction. In the recurrent net, multiple

neurons in a layer are interconnected to organize the network. In the following, we give typical characteristics of the feedforward net and the recurrent net, respectively.

5.3.1 Feedforward Networks

A feedforward network consists of a set of *input terminals* which feed the input patterns to a layer or subgroup of neurons. The layer of neurons makes independent computations on data that it receives, and passes the results to another layer. The next layer may, in turn, make its independent computations and pass on the results to yet another layer. Finally, a subgroup of one or more neurons determines the output from the network. This last layer of the network is the *output layer*. The layers that are placed between the input terminals and the output layer are called *hidden layers*.

Some authors refer to the input terminals as the input layer of the network. We do not use that convention since we wish to avoid ambiguity. Note that each neuron in a network makes its computation based on the weighted sum of its inputs. There is one exception to this rule: the role of the 'input layer' is somewhat different as units in this layer are used only to hold input data, and to distribute the data to units in the next layer. Thus, the 'input layer' units perform no function—other than serving as a buffer, fanning out the inputs to the next layer. These units do not perform any computation on the input data, and their weights, strictly speaking, do not exist.

The network outputs are generated from the output layer units. The output layer makes the network information available to the outside world. The hidden layers are internal to the network and have no direct contact with the external environment. There may be from zero to several hidden layers. The network is said to be *fully connected* if every output from a single node is channeled to every node in the next layer.

The number of input and output nodes needed for a network will depend on the nature of the data presented to the network, and the type of the output desired from it, respectively. The number of neurons to use in a hidden layer, and the number of hidden layers required for processing a task, is less obvious. Further comments on this question will appear later.

A Layer of Neurons

A one-layer network with n inputs and M neurons is shown if Fig. 5.9. In the network, each input x_j; $j = 1, 2, \ldots, n$ is connected to the qth neuron input through the weight w_{qj}; $q = 1, 2, \ldots, M$. The qth neuron has a *summer* that gathers its weighted inputs to form its own scalar output

$$\sum_{j=1}^{n} w_{qj}\, x_j + w_{q0};\ q = 1, 2, \ldots, M$$

Finally, the qth neuron outputs $\hat{y}_q$ through its activation function $\sigma(\cdot)$:

$$\hat{y}_q = \sigma\left(\sum_{j=1}^{n} w_{qj}\, x_j + w_{q0} \right);\ q = 1, 2, \ldots, M \tag{5.10a}$$

$$= \sigma(\mathbf{w}_q^T \mathbf{x} + w_{q0});\ q = 1, 2, \ldots, M \tag{5.10b}$$

where weight vector $\mathbf{w}_q$ is defined as,

$$\mathbf{w}_q^T = [w_{q1}\ w_{q2}\ \ldots\ w_{qn}] \tag{5.10c}$$

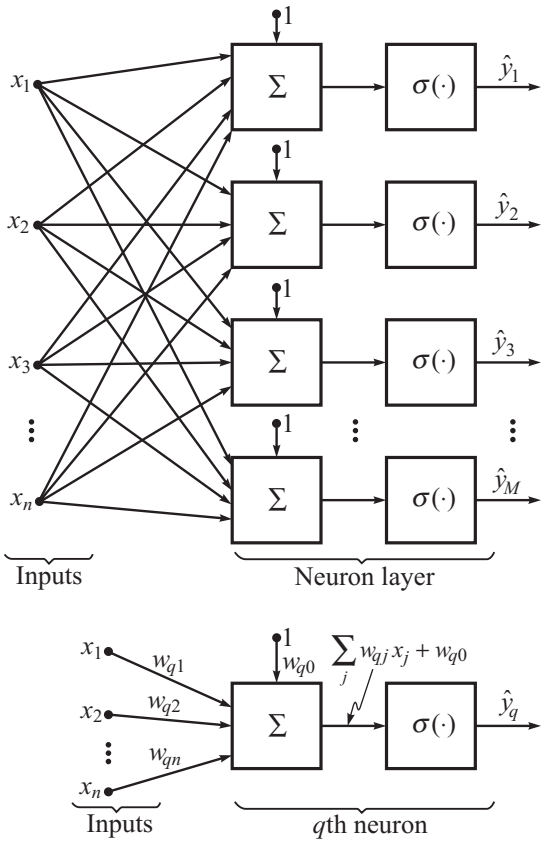


Figure 5.9 A one-layer network

Note that it is common for the number of inputs to be different from the number of neurons (i.e., $n \neq M$). A layer is not constrained to have the number of its inputs equal to the number of its neurons.

The layer shown in Fig. 5.9 has $M \times 1$ output vector

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_M \end{bmatrix}, \tag{5.11a}$$

$n \times 1$ input vector

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{5.11b}$$

$M \times n$ weight matrix

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & & \vdots \\ w_{M1} & w_{M2} & \cdots & w_{Mn} \end{bmatrix} = \begin{bmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ \vdots \\ \mathbf{w}_M^T \end{bmatrix} \tag{5.11c}$$

and $M \times 1$ bias vector

$$\mathbf{w}_0 = \begin{bmatrix} w_{10} \\ w_{20} \\ \vdots \\ w_{M0} \end{bmatrix} \tag{5.11d}$$

Note that the row indices on the elements of matrix $\mathbf{W}$ indicate the *destination* neuron for the weight, and the column indices indicate which *source* is the input for that weight. Thus, the index w_{qj} says that the signal from jth input is connected to the qth neuron.

The activation vector is,

$$\mathbf{W}\mathbf{x} + \mathbf{w}_0 = \begin{bmatrix} \mathbf{w}_1^T \mathbf{x} + w_{10} \\ \mathbf{w}_2^T \mathbf{x} + w_{20} \\ \vdots \\ \mathbf{w}_M^T \mathbf{x} + w_{M0} \end{bmatrix} \tag{5.11e}$$

The outputs are,

$$\begin{aligned} \hat{y}_1 &= \sigma(\mathbf{w}_1^T \mathbf{x} + w_{10}) \\ \hat{y}_2 &= \sigma(\mathbf{w}_2^T \mathbf{x} + w_{20}) \\ &\vdots \\ \hat{y}_M &= \sigma(\mathbf{w}_M^T \mathbf{x} + w_{M0}) \end{aligned} \tag{5.11f}$$

The input-output mapping is of the feedforward and instantaneous type since it involves no time delay between the input $\mathbf{x}$ and the output $\hat{\mathbf{y}}$.

Consider a neural network with a single output node. For a dataset with n attributes, the output node receives $x_1, x_2, \ldots, x_n$, takes a weighted sum of these and applies the $\sigma(\cdot)$ function. The output of the neural network is therefore $\sigma\left(\sum_{j=1}^{n} w_j x_j + w_0\right)$.

First consider a numerical output y (i.e., $y \in \Re$). If $\sigma(\cdot)$ is a linear activation function (Eqn (5.8)), the output is simply

$$\hat{y} = \sum_{j=1}^{n} w_j x_j + w_0$$

This is exactly equivalent to the formulation of linear regression given earlier in Section 3.6 (refer to Eqn (3.70)).

Now consider binary output variable y. If $\sigma(\cdot)$ is log-sigmoid function (Eqn (5.7a)), the output is simply

$$\hat{y} = \frac{1}{1 + \exp\left(- \sum_{j=1}^{n} w_j x_j + w_0 \right)}$$

$$= \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + w_0)}}$$

which is equivalent to logistic regression formulation given in Section 3.7 (refer to Eqn (3.84)). Note that here $\hat{y}$ takes continuous values in the interval $\{0, 1\}$ and represents the probability of belonging to Class q, i.e., $\hat{y} = P(\text{Class } 1|\mathbf{x})$, and $P(\text{Class } 2|\mathbf{x}) = 1 - \hat{y}$.

In both cases, although the neural network models are equivalent to the linear and logistic regression models, the resulting estimates for the weights in neural network models will be different from those in linear and logistic regression. This is because the estimation methods are different. As we will shortly see, the neural network estimation method is different from maximum likelihood method used in logistic regression, and may be different from least-squares method used in linear regression.

We will use multiple output nodes $\hat{y}_q$; $q = 1, \ldots, M$, for multiclass discrimination problems (detailed in Section 5.8). For regression (function approximation) problems, multiple output nodes correspond to multiple response variables we are interested in for numeric prediction. In this case, a number of regression problems are learned at the same time. An alternative is to train separate networks for separate regression problems (with one output node). In this chapter, we will focus on this alternative approach. Our focus is justified on the ground that in many real-life applications, we are interested in only one response variable, i.e., scalar output variable.

Multi-Layer Perceptrons

Neural networks normally have at least two layers of neurons, with the first layer neurons having nonlinear and differentiable activation functions. Such networks, as we will see, can approximate any nonlinear function. In real life, we are faced with nonlinear problems, and multilayer neural network structures have the capability of providing solutions to these problems.

Figure 5.10 shows a two-layer NN, with n inputs and two layers of neurons. The first of these layers has m neurons feeding into the second layer possessing M neurons. The first layer or the *hidden layer*, has *m hidden-layer neurons*; the second or the output layer, has *M output-layer neurons*. It is not uncommon for different layers to have different numbers of neurons. The outputs of the hidden layer are inputs to the following layer (output layer); and the network is fully connected. Neural

networks possessing several layers are known as *Multi-Layer Perceptrons* (MLP); their computing power is meaningfully improved over the one-layer NN.

All continuous functions, which display certain smoothness, can be approximated to any desired accuracy with a network of one hidden layer of sigmoidal hidden units, and a layer of linear output units [83]. Does this mean that it is not required to employ more than one hidden layer and/or mix different kinds of activation functions? In fact, the accuracy may be enhanced with the help of network architectures with more hidden layers/mixing activation functions. Especially when the mapping to be learned is highly complicated, there is a likelihood of performance improvement. However, as the implementation and training of the network become increasingly complex with sophisticated network architectures, it is normal to apply only a single hidden layer of similar activation functions, and an output layer of linear units. We will focus on two-layer feedforward neural networks with sigmoidal/hyperbolic tangent hidden units and linear output units for function approximation problems. For classification problems, the linear output units will be replaced with sigmoidal units. These are widely used network architectures, and work very well in many practical applications.

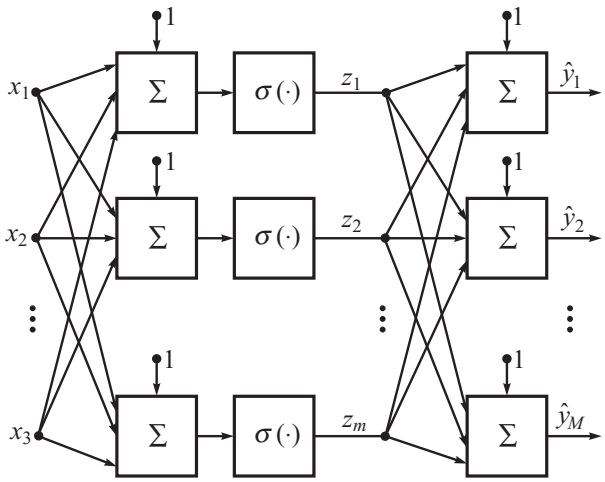


Figure 5.10 A two-layer network

Defining the input terminals as x_j; $j = 1, \ldots, n$; and the hidder-layer outputs as z_l, allows one to write

$$z_l = \sigma\left(\sum_{j=1}^{n} w_{lj} x_j + w_{l0}\right); \; l = 1, 2, \ldots, m$$

$$= \sigma(\mathbf{w}_l^T \mathbf{x} + w_{l0}) \tag{5.12a}$$

where

$$\mathbf{w}_l^T \triangleq [w_{l1} \; w_{l2} \; \ldots \; w_{ln}]$$

are the weights connecting input terminals to hidden layer.

Defining the output-layer nodes as $\hat{y}_q$, one may write the *NN* output as,

$$\hat{y}_q = \left(\sum_{l=1}^{m} v_{ql}\, z_l + v_{q0} \right);\ q = 1, \ldots, M$$
$$= \mathbf{v}_q^T \mathbf{z} + v_{q0} \tag{5.12b}$$

where

$$\mathbf{v}_q^T \triangleq [v_{q1}\ v_{q2}\ \ldots\ v_{qm}]$$

are the weights connecting hidden layer to output layer.

For the multiclass discrimination problems, our focus will be on two-layer feedforward neural networks with sigmoidal/hyperbolic tangent hidden units (outputs of hidden units given by (5.12a)), and sigmoidal output units. The *NN* output of this multilayer structure may be written as,

$$\hat{y}_q = \sigma\left(\sum_{l=1}^{m} v_{ql} z_l + v_{q0} \right);\ q = 1, \ldots, M \tag{5.12c}$$

The inputs to the output-layer units (refer to Eqns (5.12b)-(5.12c)) are the nonlinear basis function values z_l; $l = 1, \ldots, m$, computed by the hidden units. It can be said that the hidden units make a nonlinear transformation from the n-dimensional input space to the m-dimensional space spanned by the hidden units and in this space, the output layer implements a linear/logistic function.

5.3.2 Recurrent Networks

The feedforward networks (Figs 5.9–5.10) implement fixed-weight mappings from the input space to the output space. Because the networks have fixed weights, the *state* of any neuron is solely determined by the input to the unit, and not the initial and past states of the neurons. This independence of initial and past states of the network neurons limits the use of such networks because no *dynamics* are involved. The maps implemented by the feedforward networks of the type shown in Figs 5.9–5.10, are *static* maps.

To allow initial and past state involvement along with serial processing, *recurrent neural networks* utilize *feedback*. Recurrent neural networks are also characterized by use of nonlinear processing units; thus, such networks are nonlinear dynamic systems (Networks of the form shown in Figs 5.9–5.10 are nonlinear static systems).

The architectural layout of a recurrent network takes diverse forms. Feedback may come from the output neurons of a feedforward network to the input terminals. Feedback may also come from the hidden neurons of the network to the input terminals. In case the feedforward network possesses two or more hidden layers, the likely forms of feedback expand further. Recurrent networks possess a rich collection of architectural layouts.

It often turns out that several real-world problems, which are thought to be solvable only through recurrent architectures, are solvable with feedforward architectures also. A multilayer feedforward network, which realizes a static map, is capable of representing the input/output behavior of a dynamic system. To make this possible, the neural network has to be provided with information

regarding the system history—delayed inputs and outputs (refer to Section 1.4.1). The amount of history required is dependent on the level of accuracy sought, and the resulting computational complexity. Large number of inputs increase the number of weights in the network that may result in higher accuracy, but then it may significantly increase the training time. Trial-and-error on the number of inputs, as well as the network structures, is the search process as in other machine learning systems (Later sections will give more details).

From several practical applications published over the past decade, there seems to be considerable evidence that multilayer feedforward networks have an extraordinary capability to do quite well in most cases.

We will focus on two-layer feedforward neural networks with sigmoidal or hyperbolic tangent hidden units and linear/sigmoidal output units. This, in all likelihood, is the most popular network architecture as it works well in many practical applications.

The rest of this chapter is organized as follows: We first consider principles of design for the primitive units that make up artificial neural networks (perceptrons, linear units, and sigmoid units), along with learning algorithms for training single units. We then present the BACKPROPAGATION algorithm for training multilayer networks of such units, and several general issues related to the algorithm. We conclude the chapter with our discussion on RBF networks.

5.4 PERCEPTRONS

Classical NN systems are based on units called PERCEPTRON and ADALINE (ADAptive Linear Element). Perceptron was developed in 1958 by Frank Rosenblatt, a researcher in neurophysiology, to perform a kind of pattern recognition tasks. In mathematical terms, it resulted from the solution of classification problem. ADALINE was developed by Bernard Widrow and Marcian Hoff; it originated from the field of signal processing, or more specifically from the adaptive noise cancellation problem. It resulted from the solution of the regression problem; the regressor having the properties of noise canceller (linear filter).

The perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs; then outputs +1 if the result is greater than the threshold and –1 otherwise (refer to Fig. 4.2).

The ADALINE in its early stage consisted of a neuron with a linear activation function (Eqn 5.8), a hard limiter (a thresholding device with a signum activation function) and the Least Mean Square (LMS) learning algorithm. We focus on the two most important parts of ADALINE—its linear activation function and the LMS learning rule. The hard limiter is omitted, not because it is irrelevant, but for being of lesser importance to the problems to be solved. The words ADALINE and *linear neuron* are both used here for a neural processing unit with a linear activation function and a corresponding learning rule (not necessarily LMS).

The roots of both the perceptron and the ADALINE were in the linear domain. However, in real life, we are faced with nonlinear problems, and the perceptron was superseded by more sophisticated and powerful neuron and neural network structures (multilayer neural networks). What is the type of unit to be used to construct multilayer networks? Firstly, we may be encouraged to select the linear units. But, multiple layers of cascaded linear units still produce only linear functions, and we prefer networks capable of representing highly nonlinear functions. Another likely selection

could be perceptron unit. However, because of its discontinuous threshold, it is not differentiable, and therefore, not suited to the gradient descent approach for optimizing the performance criterion. What is required is a unit with output , which is a nonlinear function of its inputs—an output which is also a differentiable function of its inputs. One solution is the *sigmoid unit*, a unit similar to perceptron, but based on a smoothened, differentiable threshold function (Fig. 5.8; Eqns (5.7)). These activation functions are nothing but *softer* versions of original perceptron's hard-limiting threshold functions. In literature, these softer versions are also referred to as perceptrons, and the multilayer neural networks are also referred to as *Multi-Layer Perceptron (MLP) Networks*.

In the following, we discuss principles of perceptron learning for classification tasks. The next section gives the principles for linear-neuron learning. There after principles of 'soft' perceptron (sigmoid unit) learning will be presented.

5.4.1 Limitations of Perceptron Algorithm for Linear Classification Tasks

The roots of the Rosenblatt's perceptron were in the linear domain. It was developed as the simplest yet powerful classifier providing the *linear separability* of class patterns or examples. In Section 4.3, we have presented a detailed account of *perceptron algorithm*. It was observed that there is a major problem associated with this algorithm for real-world solutions: datasets are almost certainly *not* linearly separable, while the algorithm finds a separating hyperplane only for linearly separable data. When the dataset is linearly inseparable, the test of the decision surface will always fail for some subset of training points regardless of the adjustments we make to the free parameters, and the algorithm will loop forever. So, an upperbound needs to be imposed on the number of iterations. Thus, when perceptron algorithm is applied in practice, we have to live with the errors—true outputs will not always be equal to the desired ones.

History has proved that limitations of Rosenblatt's perceptron can be overcome by neural networks. The perceptron criterion function is based on *misclassification error* (number of samples misclassified) and the gradient procedures for minimization are not applicable. The neural networks primarily solve the regression problems, are based on *minimum squared-error criterion* (Eqn (3.71)) and employ gradient procedures for minimization. The algorithms for separable—as well as inseparable—data classification are first developed in the context of regression problems and then adapted for classification problems. Some methods for minimization of squared-error criterion were discussed in Section 3.6; the gradient procedures will be discussed in the present chapter.

5.4.2 Linear Classification using Regression Techniques

In the following, we present basic concepts of *linear* classification using regression techniques employing classical hard-limiting perceptrons.

In real-life, we are faced with *nonlinear* classification problems. The perceptron was superseded by more sophisticated and powerful neuron and neural network structures. A popular network used today—the *multilayer network*, has hidden layers of neurons with sigmoidal activations (discussed in later sections). These activation functions are nothing but *softer* versions of original perceptron's hard-limiting threshold functions. In literature, these softer versions are also referred to as perceptrons. Using a Multi-Layer Perceptron (MLP) network for nonlinear classification is not radically different; it directly follows from the concepts for linear classification discussed below.

The regression techniques discussed in Section 3.6, and also later in this chapter, can be used for linear classification with a careful choice of the target values associated with classes. Let the set of training (data) examples $\mathcal{D}$ be

$$\begin{aligned}\mathcal{D} &= \{x_j^{(i)}, y^{(i)}\};\ i = 1, \ldots, N; j = 1, \ldots, n \\ &= \{\mathbf{x}^{(i)}, y^{(i)}\}\end{aligned} \tag{5.13}$$

where $\mathbf{x}^{(i)} = [x_1^{(i)}\ x_2^{(i)}\ \ldots\ x_n^{(i)}]^T$ is an n-dimensional *input vector* (pattern with n-features) for the ith example in a real-valued space; $y^{(i)}$ is its *class label* (output value), and $y^{(i)} \in [+1, -1]$, +1 denotes Class 1 and –1 denotes Class 2. To build a linear classifier, we need a linear function of the form

$$g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0 \tag{5.14}$$

so that the input vector $\mathbf{x}^{(i)}$ is assigned to Class 1 if $g(\mathbf{x}^{(i)}) > 0$, and to Class 2 if $g(\mathbf{x}^{(i)}) < 0$, i.e.,

$$\hat{y}^{(i)} = \begin{cases} +1 \text{ if } \mathbf{w}^T\mathbf{x}^{(i)} + w_0 > 0 \\ -1 \text{ if } \mathbf{w}^T\mathbf{x}^{(i)} + w_0 < 0 \end{cases} \tag{5.15}$$

$\mathbf{w} = [w_1\ w_2\ \ldots w_n]^T$ is the *weight vector* and w_0 is the *bias*. In terms of regression, we can view this classification problem as follows.

Given a vector $\mathbf{x}^{(i)}$, the output of the *summing unit* (linear combiner) will be $\mathbf{w}^T\mathbf{x}^{(i)} + w_0$ (decision hyperplane) and thresholding the output through a *sgn* function gives us perceptron output $\hat{y}^{(i)} = \pm 1$ (refer to Fig. 5.11).

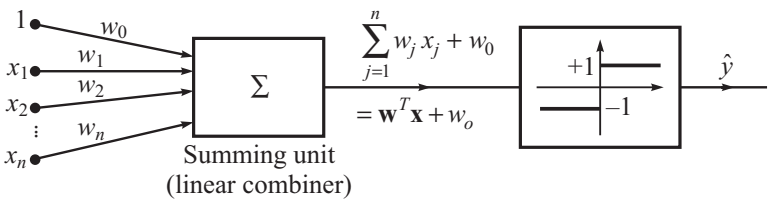


Figure 5.11 Linear classification using regression technique

The sum of error squares for the classifier becomes (Eqns (3.71))

$$E = \sum_{i=1}^{N} (e^{(i)})^2 = \sum_{i=1}^{N} (y^{(i)} - \hat{y}^{(i)})^2 \tag{5.16}$$

We require E to be a function of $(\mathbf{w}, w_0)$ to design the linear function $g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0$ that minimizes E. To obtain $E(\mathbf{w}, w_0)$, we replace the perceptron outputs $\hat{y}^{(i)}$ by the linear combiner outputs $\mathbf{w}^T\mathbf{x}^{(i)} + w_0$; this gives us the error function

$$E(\mathbf{w}, w_0) = \sum_{i=1}^{N} (y^{(i)} - (\mathbf{w}^T\mathbf{x}^{(i)} + w_0))^2 \tag{5.17}$$

Consider pattern i. If $\mathbf{x}^{(i)} \in$ Class 1, the desired output $y^{(i)} = +1$ with summing unit output $\mathbf{w}^T\mathbf{x}^{(i)} + w_0 > 0$ ($\hat{y}^{(i)} = +1$, refer to Fig. 5.11), the contribution of correctly classified pattern i to $E(\mathbf{w}, w_0)$ is small when compared with wrongly classified pattern ($\mathbf{w}^T\mathbf{x}^{(i)} + w_0 < 0$; $\hat{y}^{(i)} = -1$).

The error function $E(\mathbf{w}, w_0)$ in Eqn (5.17), is a continuous, differentiable function; therefore, gradient descent approach (discussed later in this sub-section) for minimization of $E(\mathbf{w}, w_0)$ will be applicable. The training algorithm based on this $E(\mathbf{w}, w_0)$ can be seen as the training algorithm of a *linear neuron* without the nonlinear (signum) activation function. Nonlinearity is ignored during training; after training and once the weights have been fixed, the model is the perceptron model with the hard limiter following the linear combiner.

- If the unthresholded output $\mathbf{w}^T\mathbf{x}^{(i)} + w_0$ can be trained to fit the desired values $y^{(i)} = \pm 1$ in a perfect way, then the thresholded output will fit them as well (because *sgn* (1) = 1 and *sgn* (– 1) = –1). Even when the target values cannot fit perfectly in the unthresholded case, the thresholded value will correctly fit the ±1 target value whenever the unthresholded output has the correct sign. Note, however, that while gradient descent procedure will learn weights that minimize the error in the unthresholded output, these weights will not necessarily minimize the number of training examples misclassified by the thresholded output.
- The perceptron training rule (Section 4.3) converges after a finite number of iterations to a weight vector that perfectly classifies the training data provided the training examples are linearly separable. The gradient descent rule converges only asymptotically toward the minimum-error weight vector, possibly requiring unbounded time, but converges regardless of whether the training data is linearly separable or not.

5.4.3 Standard Gradient Descent Optimization Scheme: Steepest Descent

Gradient descent serves as the basis for learning algorithms that search the hypothesis space of possible weight vectors to find the weights that best fit the training examples. The gradient descent training rule for a single neuron is important because it provides the basis for the BACKPROPAGATION algorithm, which can learn networks with many interconnected units.

For linear classification using regression techniques, the task is to train unthresholed perceptron (it corresponds to the first stage of perceptron, without the threshold; Fig. 5.11) for which the output is given by

$$g(\mathbf{x}) = \sum_{j=1}^{n} w_j x_j + w_0 = \mathbf{w}^T\mathbf{x} + w_0 \tag{5.18}$$

Let us define a single weight vector $\bar{\mathbf{w}}$ for the weights ($\mathbf{w}$, w_0):

$$\bar{\mathbf{w}}^T = [w_0\ w_1\ w_2\ \ldots\ w_n]^T \tag{5.19}$$

In terms of the weight vector $\bar{\mathbf{w}}$, the output

$$g(\mathbf{x}) = \bar{\mathbf{w}}^T\bar{\mathbf{x}} \tag{5.20a}$$

where

$$\bar{\mathbf{x}}^T = [x_0\ x_1\ x_2\ \ldots\ x_n]^T;\ x_0 = 1 \tag{5.20b}$$

The unthresholded output $\bar{\mathbf{w}}^T \bar{\mathbf{x}}^{(i)}$ is to be trained to fit the desired values $y^{(i)}$ minimizing the error (Eqn (5.17))

$$E(\bar{\mathbf{w}}) = \tfrac{1}{2} \sum_{i=1}^{N} (y^{(i)} - \bar{\mathbf{w}}^T \bar{\mathbf{x}}^{(i)})^2 \tag{5.21}$$

This error function is a continuous, differentiable function; therefore, gradient descent approach for minimization of $E(\bar{\mathbf{w}})$ will be applicable (the constant $\frac{1}{2}$ is used for computational convenience only; it gets cancelled out by the differentiation required in the error minimization process).

To understand the gradient descent algorithm, it is helpful to visualize the error space of possible weight vectors and the associated values of the *performance criterion* (*cost function E*). For the unthresholded unit (a *linear* weighted combination of inputs), the error surface is parabolic with a single global minimum. The specific parabola will depend, of course, on the particular set of training examples.

How can we calculate the direction of steepest descent along the error surface? This direction can be found by computing the derivative of *E* with respect to each component of the vector $\bar{\mathbf{w}}$. This vector-derivative is called the *gradient* of *E* with respect to $\bar{\mathbf{w}}$, written $\nabla E(\bar{\mathbf{w}})$.

$$\nabla E(\bar{\mathbf{w}}) = \left[\frac{\partial E}{\partial w_0} \; \frac{\partial E}{\partial w_1} \ldots \frac{\partial E}{\partial w_n} \right]^T \tag{5.22}$$

When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in E. The negative of this vector, therefore, gives the direction of *steepest decrease*. Therefore, the training rule for gradient descent is,

$$\bar{\mathbf{w}} \leftarrow \bar{\mathbf{w}} + \Delta \bar{\mathbf{w}} \tag{5.23a}$$

where

$$\Delta \bar{\mathbf{w}} = -\eta \nabla E(\bar{\mathbf{w}}) \tag{5.23b}$$

Here η is a positive constant (less than one), called the *learning rate* which determines the step size in the gradient descent search. This training rule can also be written in its component form:

$$w_j \leftarrow w_j + \Delta w_j; j = 0, 1, 2, \ldots, n \tag{5.24a}$$

where

$$\Delta w_j = -\eta \frac{\partial E}{\partial w_j} \tag{5.24b}$$

which shows that steepest descent is achieved by altering each component w_j of $\bar{\mathbf{w}}$ in proportion to $\frac{\partial E}{\partial w_j}$.

Gradient descent search helps determine a weight vector that minimizes *E* by starting with an arbitrary initial weight vector and then altering it again and again in small steps. At each step, the weight vector is changed in the direction producing the steepest descent along the error surface. The process goes on till the global minimum error is attained.

To build a practical algorithm for repeated updation of weight according to (5.24), we require an effective technique to calculate the gradient at each step. Luckily, this is quite easy. The gradient with respect to weight w_j; $j = 1, \ldots, n$, can be obtained by differentiating *E* from Eqn (5.21) as,

$$\frac{\partial E}{\partial w_j} = \frac{\partial}{\partial w_j}\left[\frac{1}{2}\sum_{i=1}^{N}\left(y^{(i)} - \left(\sum_{j=1}^{n} w_j x_j^{(i)} + w_0\right)\right)^2\right]$$

The error $e^{(i)}$ for the i^{th} sample of data is given by $e^{(i)} = y^{(i)} - \left(\sum_{j=1}^{n} w_j x_j^{(i)} + w_0\right)$.
It follows that

$$\frac{1}{2}\sum_{i=1}^{N}\left[\frac{\partial}{\partial w_j}(e^{(i)})^2\right] = \sum_{i=1}^{N} e^{(i)}\,\frac{\partial e^{(i)}}{\partial w_j}$$

$$\frac{\partial e^{(i)}}{\partial w_j} = -x_j^{(i)}$$

$$\frac{\partial E}{\partial w_j} = \frac{\partial}{\partial w_j}\left[\frac{1}{2}\sum_{i=1}^{N}(e^{(i)})^2\right]$$

$$= -\sum_{i=1}^{N} e^{(i)} x_j^{(i)}$$

$$= -\sum_{i=1}^{N}\left(y^{(i)} - \left(\sum_{j=1}^{n} w_j x_j^{(i)} + w_0\right)\right)x_j^{(i)}$$

The gradient with respect to bias,

$$\frac{\partial E}{\partial w_0} = -\sum_{i=1}^{N} e^{(i)} = -\sum_{i=1}^{N}\left(y^{(i)} - \left(\sum_{j=1}^{n} w_j x_j^{(i)} + w_0\right)\right)$$

Therefore, the weight update rule for gradient descent becomes

$$w_j \leftarrow w_j + \eta \sum_{i=1}^{N}\left(y^{(i)} - \left(\sum_{j=1}^{n} w_j x_j^{(i)} + w_0\right)\right)x_j^{(i)} \tag{5.25a}$$

$$w_0 \leftarrow w_0 + \eta \sum_{i=1}^{N}\left(y^{(i)} - \left(\sum_{j=1}^{n} w_j x_j^{(i)} + w_0\right)\right) \tag{5.25b}$$

An *epoch* is a complete run through all the N associated pairs. Once an epoch is completed, the pair $(\mathbf{x}^{(1)}, y^{(1)})$ is presented again and a run is performed through all the pairs again. After several epochs, the ouput error is expected to be sufficiently small.

The iteration index k corresponds to the number of times the set of N pairs is presented and cumulative error is compounded. That is, k corresponds to the epoch number.

In terms of iteration index k, the weight update equations are

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \eta \sum_{i=1}^{N} \left(y^{(i)} - \left(\sum_{j=1}^{n} w_j(k) x_j^{(i)} + w_0(k) \right) \right) \mathbf{x}^{(i)} \tag{5.26a}$$

$$w_0(k+1) = w_0(k) + \eta \sum_{i=1}^{N} \left(y^{(i)} - \left(\sum_{j=1}^{n} w_j(k) x_j^{(i)} + w_0(k) \right) \right) \tag{5.26b}$$

5.5 LINEAR NEURON AND THE WIDROW-HOFF LEARNING RULE

The perceptron ('softer' version; sigmoid unit) has been a fundamental building block in the present-day neural models. Another important building block has been ADALINE (ADAptive LINear Element), developed by Bernard Widrow and Marcian Hoff in 1959. It originated from the field of signal processing, or more specifically, from the adaptive noise cancellation problem. It resulted from the solution of regression problem; the regressor having the properties of noise canceller (linear filter). All its power in linear domain is still in full service, and despite being a simple neuron, it is present (without a thresholding device) in almost all the neural models for regression functions. The words ADALINE and *linear neuron* are both used here for a neural processing unit with a linear activation function shown in Fig. 5.12a. The neuron labeled with summation sign only (Fig. 5.12b) is equivalent to linear neuron of Fig. 5.12a.

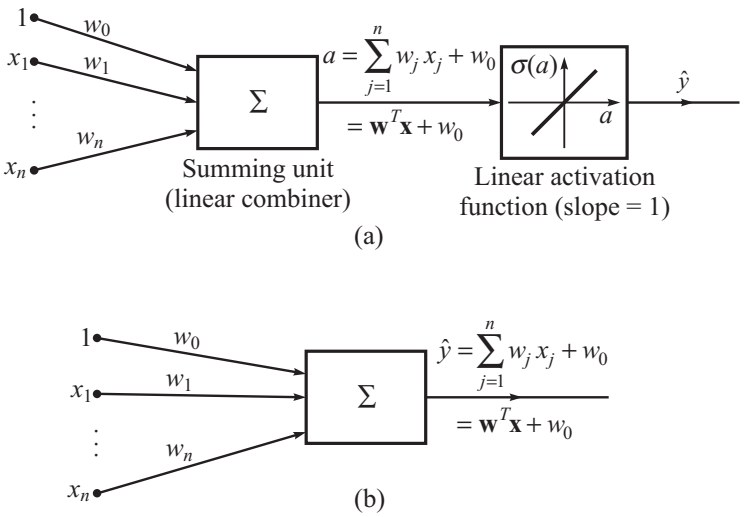


Figure 5.12 Neural processing unit with a linear activation function

In the last section, we have discussed gradient descent optimization scheme to determine the optimum setting of the weights ($\mathbf{w}$, w_0) that minimize the criterion function given by Eqn (5.21). Note that this 'sum of error squares' criterion function is deterministic and the gradient descent

scheme gives a deterministic algorithm for minimization of this function. Now we explore a digress from this criterion function. Consider the problem of computing weights ($\mathbf{w}$, w_0) so as to minimize *Mean Square Error* (MSE) between desired and true outputs, defined as follows.

$$E(\mathbf{w}, w_0) = \mathbb{E}\left[\frac{1}{2}\sum_{i=1}^{N}\left(y^{(i)} - (\mathbf{w}^T\mathbf{x}^{(i)} + w_0)\right)^2\right] \tag{5.27}$$

where $\mathbb{E}$ is the statistical expectation operator.

The solution to this problem requires the computation of autocorrelation matrix $\mathbb{E}[\mathbf{x}\,\mathbf{x}^T]$ of the set of feature vectors, and cross-correlation matrix $\mathbb{E}[\mathbf{x}y]$ between the desired response and the feature vector. This presupposes knowledge of the underlying distributions, which, in general, is not known. Thus, our major goal becomes to see if it is possible to solve this optimization problem without having this statistical information.

The *Least Mean Square* (LMS) algorithm, originally formulated by Widrow and Hoff, is a *stochastic gradient algorithm* that iterates weights ($\mathbf{w}$, w_0) in the regressor after each presentation of data sample, unlike the standard gradient descent that iterates weights after presentation of the whole training dataset. That is, the k^{th} iteration in standard gradient descent means the k^{th} *epoch*, or the k^{th} presentation of the whole training dataset, while k^{th} iteration in stochastic gradient descent means the presentation of k^{th} single training data pair (drawn in sequence or randomly). Thus, the calculation of the weight change $\Delta\bar{\mathbf{w}}$ or the gradient needed for this, is *pattern-based*, not *epoch-based* ($\Delta\bar{\mathbf{w}} = -\eta\,\nabla E(\bar{\mathbf{w}})$; Eqn (5.23b)).

LMS is called a stochastic gradient algorithm because the gradient vector is chosen at 'random' and not, as in steepest descent case, precisely derived from the shape of the total error surface. Random means here the instantaneous value of the gradient. This is then used as the estimator of the true quantity.

The design of the LMS algorithm is very simple, yet a detailed analysis of its convergence behavior is a challenging mathematical task. It turns out that under mild conditions, the solution provided by the LMS algorithm converges in probability to the solution of the sum-of-error-squares optimization problem.

5.5.1 Stochastic Gradient Descent

While the standard gradient descent training rule of Eqn (5.26) calculates weight updates after summing errors over *all* the training examples in the given dataset $\mathcal{D}$; the concept behind stochastic gradient descent is to approximate this gradient descent search by updating weights *incrementally*, following the calculation of the error for *each* individual example. This modified training rule is like the training rule given by Eqns (5.26) except that as we iterate through each training example, we update the weights according to the gradient with respect to the distinct error function,

$$E(k) = \tfrac{1}{2}[y^{(i)} - \hat{y}(k)]^2 = \tfrac{1}{2}[e(k)]^2 \tag{5.28a}$$

$$\hat{y}(k) = \sum_{j=1}^{n} w_j(k)\, x_j^{(i)} + w_0(k) \tag{5.28b}$$

where k is the iteration index. Note that the input components $x_j^{(i)}$ and the desired output $y^{(i)}$ are not functions of the iteration index. Training pairs $(\mathbf{x}^{(i)}, y^{(i)})$, drawn in sequence or randomly, are presented to the network at each iteration. The gradients with respect to weights and bias are computed as follows:

$$\frac{\partial E(k)}{\partial w_j(k)} = e(k)\frac{\partial e(k)}{\partial w_j(k)} = -e(k)\frac{\partial \hat{y}(k)}{\partial w_j(k)}$$

$$= -e(k)x_j^{(i)}$$

$$\frac{\partial E(k)}{\partial w_0(k)} = -e(k)$$

The stochastic gradient descent algorithm becomes,

$$w_j(k+1) = w_j(k) + \eta\, e(k)\, x_j^{(i)} \tag{5.29a}$$

$$w_0(k+1) = w_0(k) + \eta\, e(k) \tag{5.29b}$$

In terms of vectors, this algorithm may be expressed as,

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \eta\, e(k)\, \mathbf{x}^{(i)} \tag{5.30a}$$

$$w_0(k+1) = w_0(k) + \eta\, e(k) \tag{5.30b}$$

Stochastic gradient training algorithm iterates over the training examples $i = 1, 2, \ldots, N$ (drawn in sequence or randomly); at each iteration, altering weights as per the above equations. The sequence of these weight updates, iterated over all the training examples, gives rise to reasonable approximation to the gradient with respect to the entire set of training data. *By making the values of η small enough, stochastic gradient descent can be made to approximate standard gradient descent (steepest descent) arbitrarily closely.*

At each presentation of data $(\mathbf{x}^{(i)}, y^{(i)})$, one step of training algorithm is performed which updates both the weights and the bias. Note that teaching the network one fact at a time from one data pair, does not work. All the weights and the bias set so meticulously for one fact, could be drastically altered in learning the next fact. The network has to learn everything together, finding the best weights and bias settings for the total set of facts. Therefore, with incremental learning, the training should stop only after an epoch has been completed.

5.6 THE ERROR-CORRECTION DELTA RULE

In this section, gradient descent strategy for adapting weights for a single neuron having differentiable activation function is demonstrated. This will just be a small (nonlinear) deviation from the derivation of adaptive rule for the linear activation function, given in the previous section. Including this small deviation will be a natural step for deriving gradient-descent based algorithm for multilayer neural networks (in the next section, we will derive this algorithm).

A neural unit with any differentiable function $\sigma(a)$ is shown in Fig. 5.13. It first computes a linear combination of its inputs (activation value a); then applies nonlinear activation function $\sigma(a)$

to the result. The output $\hat{y}$ of nonlinear unit is a continuous function of its input a. More precisely, the nonlinear unit computes its output as,

$$\hat{y} = \sigma(a) \tag{5.31a}$$

$$a = \sum_{j=1}^{n} w_j x_j + w_0 \tag{5.31b}$$

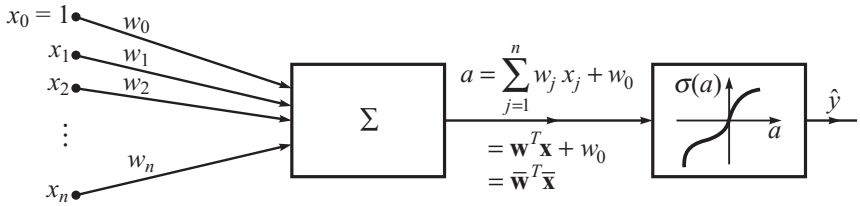


Figure 5.13 Neural unit with any differentiable activation function

The problem is to find the expression for the learning rule for adapting weights using a training set of pairs of input and output patterns; the learning is in stochastic gradient descent mode, as in the last section. We begin by defining error function $E(k)$:

$$E(k) = \tfrac{1}{2}(y^{(i)} - \hat{y}(k))^2 = \tfrac{1}{2}[e(k)]^2 \tag{5.32a}$$

$$e(k) = y^{(i)} - \hat{y}(k) \tag{5.32b}$$

$$\hat{y}(k) = \sigma\left(\sum_{j=1}^{n} w_j(k)\, x_j^{(i)} + w_0(k)\right) \tag{5.32c}$$

For each training example i, weights $w_j; j = 1, \ldots, n$ (and bias w_0) are updated by adding to it Δw_j (and Δw_0).

$$\Delta w_j(k) = -\eta \frac{\partial E(k)}{\partial w_j(k)} \tag{5.33a}$$

$$w_j(k+1) = w_j(k) - \eta \frac{\partial E(k)}{\partial w_j(k)} \tag{5.33b}$$

$$w_0(k+1) = w_0(k) - \eta \frac{\partial E(k)}{\partial w_0(k)} \tag{5.33c}$$

Note that $E(k)$ is a nonlinear function of the weights now, and the gradient cannot be calculated following the equations derived in the last section for a linear neuron. Fortunately, the calculation of the gradient is straight forward in the nonlinear case as well. For this purpose, the chain rule is,

$$\frac{\partial E(k)}{\partial w_j(k)} = \frac{\partial E(k)}{\partial a(k)} \frac{\partial a(k)}{\partial w_j(k)} \tag{5.34}$$

where the first term on the right-hand side is a measure of an *error change* due to the activation value $a(k)$ at the k^{th} iteration, and the second term shows the influence of the weights on that particular activation value $a(k)$. Applying the chain rule again, we get,

$$\frac{\partial E(k)}{\partial w_j(k)} = \frac{\partial E(k)}{\partial e(k)} \frac{\partial e(k)}{\partial \hat{y}(k)} \frac{\partial \hat{y}(k)}{\partial a(k)} \frac{\partial a(k)}{\partial w_j(k)}$$

$$= e(k)\,[-1]\,\frac{\partial \sigma(a(k))}{\partial a(k)}\,x_j^{(i)}$$

$$= -\,e(k)\;\sigma'(a(k))\;x_j^{(i)} \tag{5.35}$$

The learning rule can be written as,

$$w_j(k+1) = w_j(k) + \eta\; e(k)\;\sigma'(a(k))\;x_j^{(i)} \tag{5.36a}$$

$$w_0(k+1) = w_0(k) + \eta\; e(k)\;\sigma'(a(k)) \tag{5.36b}$$

This is the most general learning rule that is valid for a single neuron having any nonlinear and differentiable activation function and whose input is formed as a product of the pattern and weight vectors. It follows the LMS algorithm for a linear neuron presented in the last section, which was an early powerful strategy for adapting weights using data pairs only.

This rule is also known as *delta learning rule* with delta defined as,

$$\delta(k) = e(k)\;\sigma'(a(k))$$

$$= (y^{(i)} - \hat{y}(k))\;\sigma'(a(k)) \tag{5.37}$$

In terms of $\delta(k)$, the weights-update equations become

$$w_j(k+1) = w_j(k) + \eta\,\delta(k)\,x_j^{(i)} \tag{5.38a}$$

$$w_0(k+1) = w_0(k) + \eta\,\delta(k) \tag{5.38b}$$

It should be carefully noted that the $\delta(k)$ in these equations is not the *error* but the *error change* $-\dfrac{\partial E(k)}{\partial a(k)}$ due to the input $a(k)$ to the nonlinear activation function at the k^{th} iteration:

$$-\frac{\partial E(k)}{\partial a(k)} = -\frac{\partial E(k)}{\partial e(k)} \frac{\partial e(k)}{\partial \hat{y}(k)} \frac{\partial \hat{y}(k)}{\partial a(k)}$$

$$= -\,e(k)\,[-1]\;\sigma'(a(k))$$

$$= e(k)\;\sigma'(a(k)) = \delta(k) \tag{5.39}$$

Thus, $\delta(k)$ will generally not be equal to the *error* $e(k)$. We will use the term *error signal* for $\delta(k)$, keeping in mind that, in fact, it represents the error change.

In the world of neural computing, the error signal $\delta(k)$ is of highest importance. After a hiatus in the development of learning rules for multilayer networks for about 20 years, the adaptation rule based on delta rule made a breakthrough in 1986 and was named the *generalized delta learning rule*. Today, the rule is also known as the *error backpropagation learning rule* (discussed in the next section).

Interestingly, for a linear activation function (Fig. 5.12),

$$\sigma(a(k)) = a(k)$$

Therefore,

$$\sigma'(a(k)) = 1$$

and

$$\delta(k) = e(k)\ \sigma'(a(k)) = e(k) \tag{5.40}$$

That is, delta represents the error itself. Therefore, the delta rule for a linear neuron is same as the LMS learning rule presented in the previous section.

5.6.1 Sigmoid Unit: Soft-Limiting Perceptron

In the neural unit of Fig. 5.13, any nonlinear, smooth, differentiable, and preferably nondecreasing function can be used. The requirement for the activation function to be differentiable is basic for the error backpropagation algorithm. On the other hand, the requirement that a nonlinear activation function should monotonically increase is not so strong, and it is connected with the desirable property that its derivative does not change the sign.

The activation functions that are most commonly used in multilayer neural networks are the squashing sigmoidal functions. The sigmoidal unit is very much like a perceptron, but is based on smoothed differentiable threshold function.

The neural unit shown in Fig. 5.13 becomes a *sigmoidal unit* when $\sigma(\cdot)$ represents the sigmoidal nonlinearity illustrated in Fig. 5.8. The sigmoidal unit first computes a linear combination of its inputs (activation value a), and then applies a threshold to the result. The thresholded output $\hat{y} = \sigma(a)$ is a continuous function of its input. More precisely, the sigmoid unit computes its output as,

$$\hat{y} = \sigma(a) \tag{5.41a}$$

$$= \sigma\left(\sum_{j=1}^{n} w_j x_j + w_0\right) \tag{5.41b}$$

Because a sigmoid unit maps a very large input down to a small range outputs, it is often referred to as *squashing function*.

The most common squashing sigmoidal functions are *unipolar logistic function* (Fig. 5.8a, Eqn (5.7a)) and the *bipolar sigmoidal function* (related to a tangent hyperbolic; Fig. 5.8b, Eqn (5.7b)). The unipolar logistic function, henceforth referred to as *log-sigmoid*, squashes the inputs to outputs between 0 and 1, while the bipolar function, henceforth referred to as *tan-sigmoid*, squashes the inputs to the outputs between –1 and +1.

Log-sigmoid

$$\sigma(a) = \frac{1}{1 + e^{-a}} \tag{5.42a}$$

Tan-sigmoid

$$\sigma(a) = \frac{1 - e^{-a}}{1 + e^{-a}} \tag{5.42b}$$

Sigmoidal unit has the useful property that its derivative is easily expressed in terms of its output:

Log-sigmoid

$$\frac{d\sigma(a)}{da} = \frac{d}{da}\left[\frac{1}{1+e^{-a}}\right] = \frac{e^{-a}}{(1+e^{-a})^2} = \frac{1}{1+e^{-a}}\left[1 - \frac{1}{1+e^{-a}}\right]$$

$$= \sigma(a)\,[1-\sigma(a)] \tag{5.43a}$$

$$= \hat{y}\,(1-\hat{y}) \tag{5.43b}$$

Tan-sigmoid

$$\frac{d\,\sigma(a)}{da} = \frac{d}{da}\left[\frac{1-e^{-a}}{1+e^{-a}}\right] = \frac{2e^{-a}}{(1+e^{-a})^2} = \tfrac{1}{2}\left[1-\left(\frac{1-e^{-a}}{1+e^{-a}}\right)\right]\left[1+\left(\frac{1-e^{-a}}{1+e^{-a}}\right)\right]$$

$$= \tfrac{1}{2}(1-\sigma(a))\,(1+\sigma(a)) \tag{5.44a}$$

$$= \tfrac{1}{2}(1-\hat{y})\,(1+\hat{y}) \tag{5.44b}$$

As we shall see, the gradient descent learning makes use of these derivatives.

The most general learning rule that is valid for a single neuron having any nonlinear and differentiable activation function is given by Eqns(5.37–5.38). For the specific case of sigmoidal (log-sigmoid) nonlinearity, we have,

$$\sigma'(a(k)) = \frac{d}{da(k)}\,\sigma(a(k)) = \sigma(a(k))\,[1-\sigma(a(k))]$$

$$= \hat{y}(k)[1-\hat{y}(k)]$$

Therefore,

$$\delta(k) = e(k)\,\sigma'(a(k)) = (y^{(i)} - \hat{y}(k))\,\hat{y}(k)\,[1-\hat{y}(k)]$$

The weight-update equations become,

$$w_j(k+1) = w_j(k) + \eta\,\delta(k)\,x_j^{(i)} \tag{5.45a}$$

$$w_0(k+1) = w_0(k) + \eta\,\delta(k) \tag{5.45b}$$

$$\delta(k) = [y^{(i)} - \hat{y}(k)]\;\hat{y}(k)\,[1-\hat{y}(k)] \tag{5.45c}$$

We construct multilayer networks using sigmoid units (next section will describe commonly used structures). Initially we may be tempted to select the linear units discussed earlier. But, multiple layers of cascaded linear units continue to produce only linear functions and we favor networks possessing the ability to represent highly nonlinear functions. The (hard-limiting) perceptron unit is another likely selection, but its discontinuous threshold makes it undifferentiable and therefore,

not suited for gradient descent. We require a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs. There are many possible choices satisfying these requirements; sigmoid unit is the most popular choice.

5.7 MULTI-LAYER PERCEPTRON (MLP) NETWORKS AND THE ERROR-BACKPROPAGATION ALGORITHM

As noted in previous sections, single (hard-limiting) perceptron can only express linear decision surfaces, and single linear neuron can only approximate linear functions. In contrast, MLP networks trained by the backpropagation algorithm are capable of expressing a rich variety of nonlinear decision surfaces/approximating nonlinear functions. This section discusses how to learn such MLP networks using gradient descent algorithms similar to the ones described in previous sections. The backpropagation algorithm learns the weights for an MLP network, given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network outputs and the target values for these outputs.

A typical feedforward neural network is made up of a hierarchy of layers, and the neurons in the network are arranged along these layers. The external environment is connected to the network through *input terminals*, and the *output-layer neurons*. To build an artificial neural network, we must first decide how many layers of neurons are to be used and how many neurons in each layer. In other words, we must first choose the network architecture. The number of input terminals, and the number of output nodes in output layer depend on the nature of the data presented to the network, and the type of the output desired from it, respectively. For scalar-output applications, the network has a single output unit, while for vector-output applications, it has multiple output units.

A multi-layer perceptron (MLP) network is a feedforward neural network with one or more hidden layers. Each hidden layer has its own specific function. Input terminals accept input signals from the outside world and redistribute these signals to all neurons in a hidden layer. The output layer accepts a stimulus pattern from a hidden layer and establishes the output pattern of the entire network. Neurons in the hidden layers perform transformation of input attributes; the weights of the neurons represent the features in the transformed domain. These features are then used by the output layer in determining the output pattern. A hidden layer 'hides' its desired output. The training data provides the desired output of the network; that is, the desired outputs of output layer. There is no obvious way to know what the desired outputs of the hidden layers should be.

The derivation of error-backpropagation algorithm will be given here for two MLP structures shown in Figs 5.14 and 5.15. The functions $\sigma_o(\cdot)/\sigma_{oq}(\cdot)$ are the linear/log-sigmoid activation functions of the output layer, and the functions $\sigma_{hl}(\cdot)$ are the activation functions of the hidden layer (log-sigmoid or tan-sigmoid). These structures have one hidden layer only. Though more than one hidden layer in the network may provide some advantage for approximating some complex nonlinear functions (more on this in a later section), most of the practical MLP networks use one hidden layer of sigmoidal units.

We begin by considering network structure of Fig. 5.15 with log-sigmoid activation functions in the output layer. The results for structure of Fig. 5.14 with one linear output node will then easily follow.

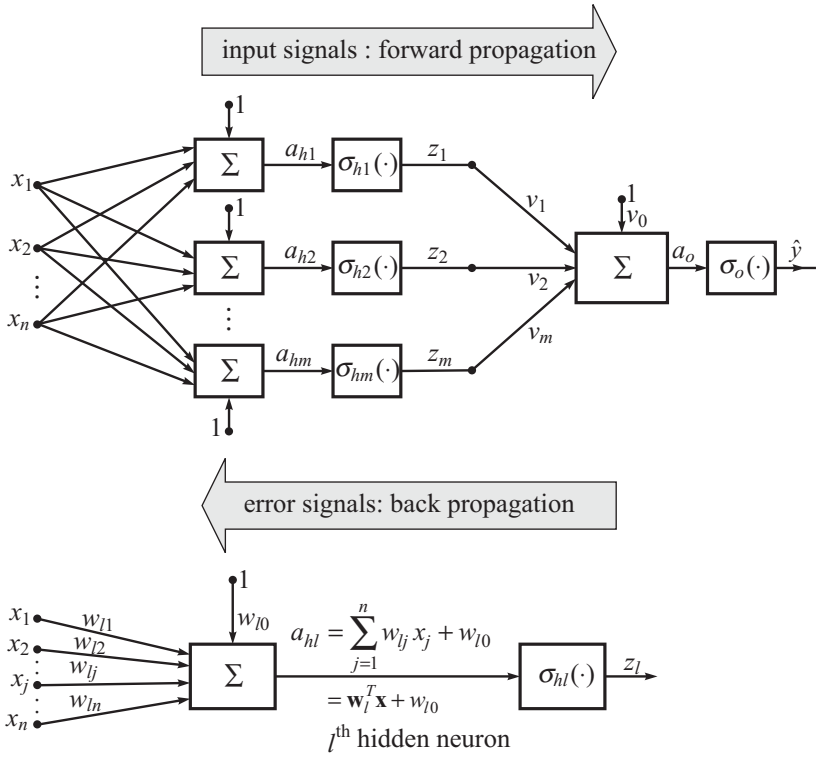


Figure 5.14 Scalar-output MLP network

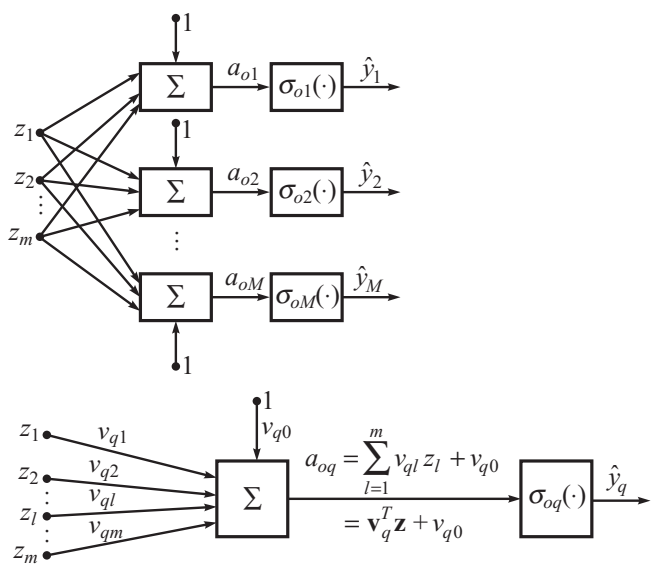


Figure 5.15 Output layer of vector-output MLP network

When **x** is fed to the input terminals (including the bias), the activation spreads in the feedforward direction, and the values z_l of the hidden units are computed (Each hidden unit is a perceptron on its own and applies the nonlinear sigmoid function to its weighted sum of inputs). The outputs $\hat{y}_q$ in the output layer are computed taking the hidden-layer outputs z_l as their inputs. Each output unit applies the log-sigmoid function to its weighted sum of inputs. It can be said that the hidden units make a nonlinear transformation from the n-dimensional input space to the m-dimensional space spanned by the hidden units, and in this space the output layer implements a log-sigmoid function.

Training Protocols

In the previous sections, we have presented two useful gradient descent optimization schemes for a single neuron: the standard gradient descent (steepest descent), and the stochastic gradient descent. In the steepest descent mode, the error function E is given by,

$$E(k) = \tfrac{1}{2}\sum_{i=1}^{N}(y^{(i)} - \hat{y}(k))^2 = \tfrac{1}{2}\sum_{i=1}^{N}[e(k)]^2 \tag{5.46a}$$

$$\hat{y}(k) = \sum_{j=1}^{n} w_j(k)\, x_j^{(i)} + w_0(k) \tag{5.46b}$$

All the patterns are presented to the network before a step of weight-update takes place.

In the stochastic gradient descent, the error function is given by,

$$E(k) = \tfrac{1}{2}(y^{(i)} - \hat{y}(k))^2 = \tfrac{1}{2}[e(k)]^2 \tag{5.47a}$$

$$\hat{y}(k) = \sum_{j=1}^{n} w_j(k)\, x_j^{(i)} + w_0(k) \tag{5.47b}$$

The weights are updated for each of the training pairs.

From now onwards, the two schemes will be referred to as two protocols of training: batch training, and incremental training[1]. Whereas the *batch training rule* computes the weight updates after summing errors over *all* the training examples (*batch of training data*), the idea behind *incremental training rule* is to update weights incrementally, following the calculation of error for *each* individual example (incremental training rule gives stochastic approximation of batch training which implements standard gradient descent (steepest descent)).

The gradient descent weight-update rules for MLP networks are similar to the *delta training rule* described in the previous section for a single neuron. The training data supplies target values $y_q^{(i)}$; $q = 1, \ldots, M$, of the outputs. Given the initial values of the weights w_{lj}; $l = 1, \ldots, m$; $j = 1, \ldots, n$, with bias weights w_{l0} between input terminals and hidden layer, and the weights v_{ql} with bias weights v_{q0} between hidden layer and output layer, the network outputs $\hat{y}_q^{(i)}$ are calculated by propagating input signals $x_j^{(i)}$ in the forward direction, computing hidden-layer outputs $z_l^{(i)}$, and therefrom the outputs $\hat{y}_q^{(i)}$. Using delta training rule, the weights v_{ql} and v_{q0} are updated with $z_l^{(i)}$ as inputs to the output neurons. The variable (or signal) δ designates an *error signal*, but not the error itself as defined in Eqn (5.45), that is, δ will generally not be equal to the error $e_q^{(i)} = y_q^{(i)} - \hat{y}_q^{(i)}$ (Interestingly, the equality does hold for linear activation function).

[1] In *on-line* training, each pattern is presented once and only once; there is no use of memory for storing the patterns. This explains the difference between incremental training and on-line training.

There is, however, a problem in updating the weights w_{lj} and w_{l0} between input terminals and the hidden layer. The weight updates of w_{lj} and w_{l0} should ideally reduce the network output error $\sum_{q=1}^{M}(y_q^{(i)} - \hat{y}_q^{(i)})$. The delta training rule requires the target values for the outputs of hidden-layer neurons, which are unknown. Hidden layer affects the network output error *indirectly* through its weights.

Since we do not know the desired values of hidden-layer outputs, the weights w_{lj} and w_{l0} could not be updated using delta training rule. After a hiatus in the development of learning rules for multilayer networks for about 20 years, the delta training rule made a breakthrough in 1986 and was named the *generalized delta rule* (today, it is more popularly known as *error-backpropagation learning rule*). The error is calculated at the output nodes, and then *propagated backwards* through the network from the output layer to the input terminals. This *error backpropagation* gives *error terms* (and not the errors themselves) for hidden units outputs. Using these error terms, the weights w_{lj} and w_{l0} between input terminals and hidden units are updated in a usual incremental/batch training mode.

Thus, in a backpropatation network, the learning algorithm has two phases: *the forward propagation to compute MLP outputs, and the back propagation to compute backpropagated errors*. In the forward propagation phase, the input signals **x** are propagated through the network from input terminals to output layer, while in the backpropagation phase, the errors at the output nodes are propagated backwards from output layer to input terminals (see Fig. 5.14). This is an indirect way in which the weights w_{lj} and w_{l0} can influence the network outputs and hence the cost function E.

The error-backpropagation algorithm (as we shall see shortly) begins by constructing a network with the desired number of hidden units and initializing the network weights to small random values. Typically, in a backpropagation network, the layers are *fully connected*, that is, every input terminal and every neuron in each layer is connected to every other neuron in the adjacent forward layer. Given this fixed network structure, the main loop of the algorithm repeatedly iterates over the training examples. For each training example, it applies the network to the example, computes the gradient with respect to the error on this example, then updates all weights in the network to implement incremental training protocol. Similar procedure is followed for implementing batch training rule.

The weight-update rule for an MLP network may be repeated thousands of times in a typical application. A range of termination conditions can be employed to stop the process. One may opt to stop after a fixed number of iterations through the loop, or once the error on the training examples drops below some threshold, or once error on a separate validation set of examples fulfills a specific criterion. The selection of proper termination criterion is important as very few iterations can be unsuccessful in reducing error sufficiently, and too many can cause overfitting of data.

5.7.1 The Generalized Delta Rule

We develop the learning rule for the case when there are multiple neurons in the output layer (Fig. 5.15). The derivation here is of learning rule for the adaptation of weights in an incremental mode.

Update Rule for Output-Units Weights

As before, we begin by defining cost function (sum-of-error-squares) for this neural network having M output-layer neurons. At each iteration step k,

$$E(k) = \tfrac{1}{2}\sum_{q=1}^{M}\left(y_q^{(i)} - \hat{y}_q(k)\right)^2 = \tfrac{1}{2}\sum_{q=1}^{M}[e_q(k)]^2 \tag{5.48a}$$

$$\hat{y}_q(k) = \sigma_{oq}\left(\sum_{l=1}^{m}[v_{ql}(k)\, z_l(k)] + v_{q0}(k)\right) \tag{5.48b}$$

$$= \sigma_{oq}(a_{oq}(k)) \tag{5.48c}$$

From forward propagation phase, we have,

$$z_l(k) = \sigma_{hl}(a_{hl}(k)) \tag{5.49a}$$

$$= \sigma_{hl}\left(\sum_{j=1}^{n}[w_{lj}(k)\, x_j^{(i)}] + w_{l0}(k)\right) \tag{5.49b}$$

We have used subscript 'o' for the output layer and subscript 'h' for the hidden layer. This is necessary because the *error signal terms* (the delta values) for output layer neurons must be distinguished from those for hidden layer processing units.

The input a_{oq} to the q^{th} output unit (Eqn (5.48b)) is given as,

$$a_{oq}(k) = \sum_{l=1}^{m}(v_{ql}(k)\, z_l(k)) + v_{q0}(k)$$

The *error signal term* for the q^{th} neuron is defined as (Eqn (5.39)),

$$\delta_{oq}(k) = -\frac{\partial E(k)}{\partial a_{oq}(k)}$$

Applying the chain rule, the gradient of the cost function with respect to the weight v_{ql} is,

$$\frac{\partial E(k)}{\partial v_{ql}(k)} = \frac{\partial E(k)}{\partial a_{oq}(k)}\frac{\partial a_{oq}(k)}{\partial v_{ql}(k)}$$

$$= -\delta_{oq}(k)\, z_l(k)$$

The weight change from Eqns (5.33) can now be written as,

$$\Delta v_{ql}(k) = -\eta\frac{\partial E(k)}{\partial v_{ql}(k)} = \eta\,\delta_{oq}(k)\, z_l(k)$$

Applying the chain rule, the expression for error signal is,

$$\delta_{oq}(k) = -\frac{\partial E(k)}{\partial a_{oq}(k)} = -\frac{\partial E(k)}{\partial \hat{y}_q(k)} \frac{\partial \hat{y}_q(k)}{\partial a_{oq}(k)}$$

$$= e_q(k)\ \sigma'_{oq}(a_{oq}(k))$$

where the term $\sigma'_{oq}(a_{oq}(k))$ represents the slope $\partial \hat{y}_q(k) / \partial a_{oq}(k)$ of the q^{th} output neuron's activation function, assumed in general, to be any nonlinear differentiable function. For our specific case of log-sigmoid output units (Eqn (5.43)),

$$\sigma'_{oq}(a_{oq}(k)) = \frac{\partial \sigma_{oq}(a_{oq}(k))}{\partial a_{oq}(k)} = \hat{y}_q(k)\,[1 - \hat{y}_q(k)]$$

resulting in a simple expression of the error signal term:

$$\delta_{oq}(k) = e_q(k)\ \hat{y}_q(k)\,[1 - \hat{y}_q(k)]$$

Finally, the weight adjustments can be calculated from

$$v_{ql}(k+1) = v_{ql}(k) + \eta\ \delta_{oq}(k)\ z_l(k) \tag{5.50a}$$

$$v_{q0}(k+1) = v_{q0}(k) + \eta\ \delta_{oq}(k) \tag{5.50b}$$

Update-Rule for Hidden-Units Weights

The problem at this point is to calculate the error signal terms δ_{hl} for the hidden-layer neurons, to update the weights w_{lj} between input terminals and hidden layer. The derivation of the expression for δ_{hl} was a major breakthrough in the learning procedure for multilayer neural networks. Unlike the output nodes, the desired outputs of the hidden nodes (and hence the errors at the hidden nodes) are unknown. If the 'target' outputs for hidden nodes were known for any input, the input terminals to hidden layer weights could be adjusted by a procedure similar to the one used for output nodes. However, there is no explicit 'supervisor' to state what the hidden units' outputs should be.

Training examples provide only the target values for the network outputs. The error signal terms δ_{oq} for the output nodes are easily calculated as we have seen.

The *generalized delta rule* provides a wayout for the computation of error signal terms for hidden units. An *intuitive* understanding of the procedure will be helpful in the derivation that follows.

To begin, notice that input terminals to hidden layer weights w_{lj} can influence the rest of the network and hence the output error only through a_{hl} (refer to Figs 5.14–5.15). The generalized delta rule computes the error signal terms δ_{hl} for hidden layer units by summing the output error signal terms δ_{oq} for each output unit influenced by a_{hl} (from Fig. 5.15, we see that a_{hl} contributes to errors at all output layer neurons), weighting each of the δ_{oq}'s by v_{ql}—the weights from the hidden unit to output units. Thus, by *error backpropagation* from output layer to the hidden layer, we take into account the *indirect* ways in which w_{lj} can influence the network outputs, and hence the error E. The power of backpropagation is that it allows us to calculate an 'effective' error for each hidden node and, thus, derive the learning rule for input terminals to hidden layer weights.

The derivation of the learning rule or of the equations for the weight change Δw_{lj} of any hidden-layer neuron follows the gradient procedure used earlier for output-layer neurons:

$$\Delta w_{lj}(k) = -\eta \frac{\partial E(k)}{\partial w_{lj}(k)} \tag{5.51a}$$

$$w_{lj}(k+1) = w_{lj}(k) - \eta \frac{\partial E(k)}{\partial w_{lj}(k)} \tag{5.51b}$$

Similar equations hold for bias weights w_{l0}.

Applying the chain rule, the gradient of the cost function with respect to the weight w_{lj} is,

$$\frac{\partial E(k)}{\partial w_{lj}(k)} = \frac{\partial E(k)}{\partial a_{hl}(k)} \frac{\partial a_{hl}(k)}{\partial w_{lj}(k)}$$

where input a_{hl} to each hidden-layer activation function is given as,

$$a_{hl}(k) = \sum_{j=1}^{n} w_{lj}(k)\, x_j^{(i)} + w_{l0}(k)$$

The error signal term for l^{th} neuron is given as,

$$\delta_{hl}(k) = -\frac{\partial E(k)}{\partial a_{hl}(k)}$$

Therefore, the gradient of the cost function becomes,

$$\frac{\partial E(k)}{\partial w_{lj}(k)} = \frac{\partial E(k)}{\partial a_{hl}(k)} \frac{\partial a_{hl}(k)}{\partial w_{lj}(k)}$$

$$= -\delta_{hl}(k)\, x_j^{(i)}$$

The weight-update equation (Eqn (5.51b)) takes the form:

$$w_{lj}(k+1) = w_{lj}(k) + \eta\, \delta_{hl}(k)\, x_j^{(i)} \tag{5.52a}$$

For the bias weights,

$$w_{l0}(k+1) = w_{l0}(k) + \eta\, \delta_{hl}(k) \tag{5.52b}$$

Now the problem in hand is to calculate the error signal term δ_{hl} for hidden-layer neuron in terms of error signal terms δ_{oq} of the output-layer neurons employing error backpropagation.

The activation a_{hl} of l^{th} hidden-layer neuron is given as,

$$a_{hl}(k) = \sum_{j=1}^{n} w_{lj}(k)\, x_j^{(i)} + w_{l0}(k)$$

The error signal term for l^{th} neuron,

$$\delta_{hl}(k) = -\frac{\partial E(k)}{\partial a_{hl}(k)}$$

Since a_{hl} contributes to errors at all output-layer neurons, we have the chain rule

$$\frac{\partial E(k)}{\partial a_{hl}(k)} = \sum_{q=1}^{M} \frac{\partial E(k)}{\partial a_{oq}(k)} \frac{\partial a_{oq}(k)}{\partial a_{hl}(k)}$$

$$= \sum_{q=1}^{M} -\delta_{oq}(k) \frac{\partial a_{oq}(k)}{\partial a_{hl}(k)}$$

$$= \sum_{q=1}^{M} -\delta_{oq}(k) \frac{\partial a_{oq}(k)}{\partial z_l(k)} \frac{\partial z_l(k)}{\partial a_{hl}(k)}$$

Since,

$$a_{oq}(k) = \sum_{l=1}^{m} v_{ql}(k)\, z_l(k) + v_{q0}$$

we have,

$$-\delta_{hl}(k) = \frac{\partial E(k)}{\partial a_{hl}(k)} = \sum_{q=1}^{M} -\delta_{oq}(k)\, v_{ql}(k)\, \sigma'_{hl}(a_{hl})$$

Assuming unipolar sigmoid activation functions in hidden-layer neurons, we have,

$$\sigma'_{hl}(a_{hl}(k)) = z_l(k)\,[1 - z_l(k)]$$

This gives,

$$\delta_{hl}(k) = z_l(k)\,[1 - z_l(k)] \sum_{q=1}^{M} \delta_{oq}(k)\, v_{ql}(k)$$

We now use the equations derived above and give the final form (to be used in algorithms) of weights update rules for the two-layer MLP networks shown in Figs (5.14 and 5.15) with unipolar sigmoidal units in the hidden layer and unipolar sigmoid/linear units in the output layer.

Weights-update equations employing incremental training for the multi-output structure (log-sigmoid output units): Our actual aim lies in learning from all the data pairs known to us. We teach the neural network with the help of one data pair at a time; weights and biases are updated after each presentation of data pair. The iteration index k corresponds to presentation of each data pair. But since the network is supposed to be learning from all the data pairs together, the stopping criterion is applied after completion of each epoch (presentation of all the N data pairs).

Forward Recursion to Compute MLP Output

Present the input $\mathbf{x}^{(i)}$ to the MLP network, and compute the output using

$$z_l(k) = \sigma_{hl}\left(\sum_{j=1}^{n}(w_{lj}(k)\, x_j^{(i)}) + w_{l0}(k)\right);\ l = 1, \ldots, m \tag{5.53a}$$

$$\hat{y}_q(k) = \sigma_{oq}\left(\sum_{l=1}^{m}(v_{ql}(k)\, z_l(k)) + v_{q0}(k))\right);\ q = 1, \ldots, M \tag{5.53b}$$

with initial weights w_{lj}, w_{l0}, v_{ql}, v_{q0}, randomly chosen.

Backward Recursion for Backpropagated Errors

$$\delta_{oq}(k) = [y_q^{(i)} - \hat{y}_q(k)]\ \hat{y}_q(k)[1 - \hat{y}_q(k)] \tag{5.53c}$$

$$\delta_{hl}(k) = z_l(k)\ [1 - z_l(k)]\ \sum_{q=1}^{M}\delta_{oq}(k)\ v_{ql}(k) \tag{5.53d}$$

Computation of MLP Weights and Bias Updates

$$v_{ql}(k+1) = v_{ql}(k) + \eta\ \delta_{oq}(k)\, z_l(k) \tag{5.53e}$$

$$v_{q0}(k+1) = v_{q0}(k) + \eta\ \delta_{oq}(k) \tag{5.53f}$$

$$w_{lj}(k+1) = w_{lj}(k) + \eta\ \delta_{hl}(k)\, x_j^{(i)} \tag{5.53g}$$

$$w_{l0}(k+1) = w_{l0}(k) + \eta\ \delta_{hl}(k) \tag{5.53h}$$

Weights-update equations employing batch training for the multi-output structure (log-sigmoid output units): Our interest lies in learning to minimize the total error over the entire batch of training examples. All N pairs are presented to the network (one at a time) and a cumulative error is computed after all pairs have been presented. At the end of this procedure, the neuron weights and biases are updated once. In batch training, the iteration index corresponds to the number of times the set of N pairs is presented and cumulative error is compounded. That is, k corresponds to epoch number.

The initial weights are randomly chosen, and iteration index k is set to 0. All the training data pairs; $i = 1, \ldots, N$, are presented to the network before the algorithm moves to iteration index $k + 1$ to update the weights. The weights-update is done using the following equations:

$$z_l^{(i)} = \sigma_{hl}\left(\sum_{l=1}^{m}(w_{lj}(k)\, x_j^{(i)}) + w_{l0}(k)\right) \tag{5.54a}$$

$$\hat{y}_q^{(i)} = \sigma_{oq}\left(\sum_{l=1}^{m}(v_{ql}(k)\, z_l^{(i)}) + v_{q0}(k)\right) \tag{5.54b}$$

$$\delta_{oq}^{(i)} = (y_q^{(i)} - \hat{y}_q^{(i)})\ \hat{y}_q^{(i)}(1 - \hat{y}_q^{(i)}) \tag{5.54c}$$

$$\Delta v_{ql}(k) = \eta \sum_{i=1}^{N}\delta_{oq}^{(i)}\ z_l^{(i)} \tag{5.54d}$$

$$\delta_{hl}^{(i)} = z_l^{(i)}(1 - z_l^{(i)}) \sum_{q=1}^{M} \delta_{oq}^{(i)} v_{ql}(k) \tag{5.54e}$$

$$\Delta w_{lj}(k) = \eta \sum_{i=1}^{N} \delta_{hl}^{(i)} x_j^{(i)} \tag{5.54f}$$

$$v_{ql}(k+1) = v_{ql}(k) + \Delta v_{ql}(k) \tag{5.54g}$$

$$w_{lj}(k+1) = w_{lj}(k) + \Delta w_{lj}(k) \tag{5.54h}$$

Weights-update equations for the multi-output structure (linear output units): Equations for multi-output structure with linear output nodes directly follow from the above equations with appropriate changes in $\sigma_{oq}(\cdot)$ and δ_{oq}.

Incremental Training

$$z_l(k) = \sigma_{hl}\left(\sum_{j=1}^{n} (w_{lj}(k)\, x_j^{(i)}) + w_{l0}(k) \right); l = 1, \ldots, m \tag{5.55a}$$

$$\hat{y}_q(k) = \sum_{l=1}^{m} (v_{ql}(k)\, z_l(k)) + v_{q0}(k)\,; q = 1, \ldots, M \tag{5.55b}$$

$$\delta_{oq}(k) = (y_q^{(i)} - \hat{y}_q(k)) \tag{5.55c}$$

$$\delta_{hl}(k) = z_l(k)\,[1 - z_l(k)] \sum_{q=1}^{M} \delta_{oq}(k)\, v_{ql}(k) \tag{5.55d}$$

$$v_{ql}(k+1) = v_{ql}(k) + \eta\, \delta_{oq}(k)\, z_l(k) \tag{5.55e}$$

$$v_{q0}(k+1) = v_{q0}(k) + \eta\, \delta_{oq}(k) \tag{5.55f}$$

$$w_{lj}(k+1) = w_{lj}(k) + \eta\, \delta_{hl}(k)\, x_j^{(i)} \tag{5.55g}$$

$$w_{l0}(k+1) = w_{l0}(k) + \eta\, \delta_{hl}(k) \tag{5.55h}$$

Batch Training

$$z_l^{(i)} = \sigma_{hl}\left(\sum_{j=1}^{n} (w_{lj}(k)\, x_j^{(i)}) + w_{l0}(k) \right) \tag{5.56a}$$

$$\hat{y}_q^{(i)} = \sum_{l=1}^{m} (v_{ql}(k)\, z_l^{(i)}) + v_{q0}(k) \tag{5.56b}$$

$$\delta_{oq}^{(i)} = (y_q^{(i)} - \hat{y}_q^{(i)}) \tag{5.56c}$$

$$\Delta v_{ql}(k) = \eta \sum_{i=1}^{N} \delta_{oq}^{(i)}\, z_l^{(i)} \tag{5.56d}$$

$$\delta_{hl}^{(i)} = z_l^{(i)}(1 - z_l^{(i)}) \sum_{q=1}^{M} \delta_{oq}^{(i)} v_{ql}(k) \tag{5.56e}$$

$$\Delta w_{lj}(k) = \eta \sum_{i=1}^{N} \delta_{hl}^{(i)} x_j^{(i)} \tag{5.56f}$$

Weights-update equations for single-output structure: Equations for single-output structure directly follow from the above equations with $M = 1$.

A summary of the error-propagation algorithm is given in Table 5.1 for the vector-output network structure, employing incremental learning.

Table 5.1 Summary of error backpropagation algorithm

Given a set of N data pairs $\{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}$; $i = 1, 2, \ldots, N$, that are used for training:

$$\mathbf{x} = [x_1\ x_2\ \ldots\ x_n]^T = \{x_j\};\ j = 1, 2, \ldots, n$$

$$\mathbf{y} = [y_1\ y_2\ \ldots\ y_M]^T = \{y_q\};\ q = 1, 2, \ldots, M$$

Forward Recursion

Step 1: Choose the number of units m in the hidden layer, the learning rate η, and predefine the maximally allowed (desired) error E_{des}.

Step 2: Initialize weights w_{lj}, w_{l0}, v_{ql}, v_{q0}; $l = 1, \ldots, m$.

Step 3: Present the input $\mathbf{x}^{(i)}$ (drawn in sequence or randomly) to the network.

Step 4: Consequently, compute the output from the hidden and output layer neurons using Eqns (5.53a–5.53b).

Step 5: Find the value of the sum of error-squares cost function $E(k)$ at the k^{th} iteration for the data pair applied and given weights (in the first step of an epoch, initialize $E(k) = 0$):

$$E(k) \leftarrow \tfrac{1}{2} \sum_{q=1}^{M} (y_q^{(i)} - \hat{y}_q(k))^2 + E(k)$$

Note that the value of the cost function is accumulated over all the data pairs.

Backward Recursion

Step 6: Calculate the error signals δ_{oq} and δ_{hl} for the output layer neurons and hidden layer neurons, respectively, using Eqns (5.53c–5.53d).

Update Weights

Step 7: Calculate the updated output layer weights $v_{ql}(k + 1)$, $v_{q0}(k + 1)$ and updated hidden layer weights $w_{lj}(k + 1)$, $w_{l0}(k + 1)$ using Eqns (5.53e–5.53h).

Contd.

Step 8: If $i < N$, go to step 3; otherwise go to step 9.

Step 9: The learning epoch (sweep through all data pairs) completed: $i = N$. For $E_N < E_{\text{des}}$, terminate training. Otherwise go to step 3 and start a new learning epoch.

Example 5.1

For sample calculations on learning by backpropagation algorithm, we consider a multilayer feedforward network shown in Fig. 5.16. Let the learning rate η be 0.9. The initial weight and bias values are given in Table 5.2. We will update the weight and bias values for one step with input vector $\mathbf{x} = [1\ 0\ 1]^T$ and desired output $y = 1$. Activation function for hidden nodes is a unipolar sigmoidal function, and for output-layer node is the linear function:

$$\sigma_{hl}(a_{hl}) = \frac{1}{1 + e^{-a_{hl}}}$$

$$\sigma_o(a_o) = a_o$$

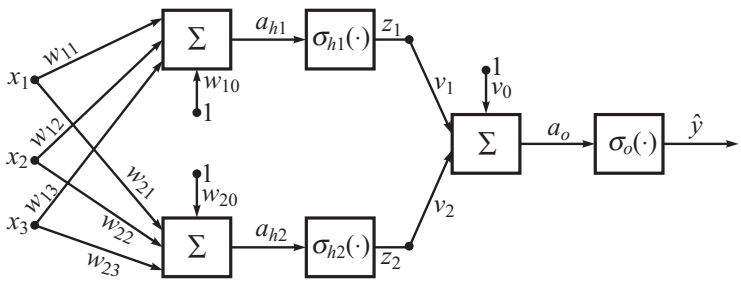


Figure 5.16 Example of a multilayer feedforward network

Table 5.2 Initial weight and bias values

Weight/Bias	w_{11}	w_{21}	w_{12}	w_{22}	w_{13}	w_{23}	w_{10}	w_{20}	v_1	v_2	v_0
Initial value	0.2	–0.3	0.4	0.1	–0.5	0.2	–0.4	0.2	–0.3	–0.2	0.1

Initial tuple $\mathbf{x} = [1\ 0\ 1]^T = [x_1\ x_2\ x_3]^T$ is presented to the network, and outputs from each hidden and output layer neurons are computed using Eqns (5.53a–5.53b):

$$a_{hl}(k) = \sum_{j=1}^{3} (w_{lj}(k)\, x_j^{(i)}) + w_{l0}(k);\ l = 1, 2$$

$$z_l(k) = \sigma_{hl}(a_{hl})$$

$$a_o(k) = v_1(k)\, z_1(k) + v_2(k)\, z_2(k) + v_0(k)$$

$$\hat{y}(k) = \sum_{l=1}^{2} (v_l(k)\, z_l(k)) + v_0(k) = a_o(k)$$

Outputs of hidden and output layer neurons are shown in Table 5.3.

The error of output unit is computed and propagated backward. The error terms are shown in Table 5.3 (Eqns (5.53c–5.53d)).

$$\delta_o(k) = y^{(i)} - \hat{y}(k)$$

$$\delta_{hl}(k) = z_l(k)\,[1 - z_l(k)]\,\delta_o(k)\,v_l(k)$$

The weight and bias updates are shown in Table 5.4 (Eqns (5.53e–5.53h)).

$$v_l(k+1) = v_l(k) + \eta(y^{(i)} - \hat{y}(k))\,z_l(k)$$

$$v_0(k+1) = v_0(k) + \eta(y^{(i)} - \hat{y}(k))$$

$$w_{lj}(k+1) = w_{lj}(k) + \eta\,z_l(k)\,[1 - z_l(k)]\,x_j^{(i)}(y^{(i)} - \hat{y}(k))\,v_l(k)$$

$$w_{l0}(k+1) = w_{l0}(k) + \eta\,z_l(k)\,[1 - z_l(k)]\,(y^{(i)} - \hat{y}(k))\,v_l(k)$$

Table 5.3 Input, output and error-term calculations for hidden and output nodes

Node	Input	Output	Error-term
Hidden 1	$a_{h1} = -0.7$	$z_1 = 0.3348$	$\delta_{h1} = -0.0735$
Hidden 2	$a_{h2} = 0.1$	$z_2 = 0.5250$	$\delta_{h2} = -0.0511$
Output	$a_o = -0.1045$	$\hat{y} = -0.1045$	$\delta_o = 1.1045$

Table 5.4 Calculations for weight and bias updates

Weight/Bias	w_{11}	w_{21}	w_{12}	w_{22}	w_{13}	w_{23}	w_{10}	w_{20}
New value	0.1339	–0.3496	0.4	0.1	–0.5661	0.1504	–0.46661	0.1504
Weight/Bias	v_1	v_2	v_0					
New value	0.0298	0.3219	1.0941					

Consider now that the activation function for hidden nodes is a unipolar sigmoid function, and for output node also, it is unipolar sigmoid. In this case,

$$a_{hl}(k) = \sum_{j=1}^{3} (w_{lj}(k)\,x_j^{(i)}) + w_{l0}(k);\; l = 1, 2$$

$$z_l(k) = \sigma_{hl}(a_{hl})$$

$$a_o(k) = v_1(k)\,z_1(k) + v_2(k)\,z_2(k) + v_0(k)$$

$$\hat{y}(k) = \sigma_o(a_o)$$

$$\delta_o(k) = (y^{(i)} - \hat{y}(k))\;\hat{y}(k)\,(1 - \hat{y}(k))$$

$$\delta_{hl}(k) = z_l(k)\,[1 - z_l(k)]\,\delta_o(k)\,v_l(k)$$
$$v_l(k+1) = v_l(k) + \eta\,\delta_o(k)\,z_l(k)$$
$$v_0(k+1) = v_0(k) + \eta\,\delta_o(k)$$
$$w_{lj}(k+1) = w_{lj}(k) + \eta\,\delta_{hl}(k)\,x_j^{(i)}$$
$$w_{l0}(k+1) = w_{l0}(k) + \eta\,\delta_{hl}(k)$$

Sample calculations are left as an exercise for the reader.

5.7.2 Convergence and Local Minima

As discussed earlier, the back propagation algorithm implements a gradient descent search through the space of possible network weights, reducing the error E between the target values and the network outputs. As the error surface for MLP networks may comprise various local minima, gradient descent could get trapped in any of these. Due to this, error back propagation over MLP networks is only assured to converge toward a local minima in E and not essentially to the global minimum error.

Let us consider, as an illustrative example, minimization of the following nonlinear function [85]:

$$E(w) = 0.5\,w^2 - 8\sin w + 7$$

where w is a *scalar* weight parameter. This function has two local minima and a global minimum (Fig. 5.17).

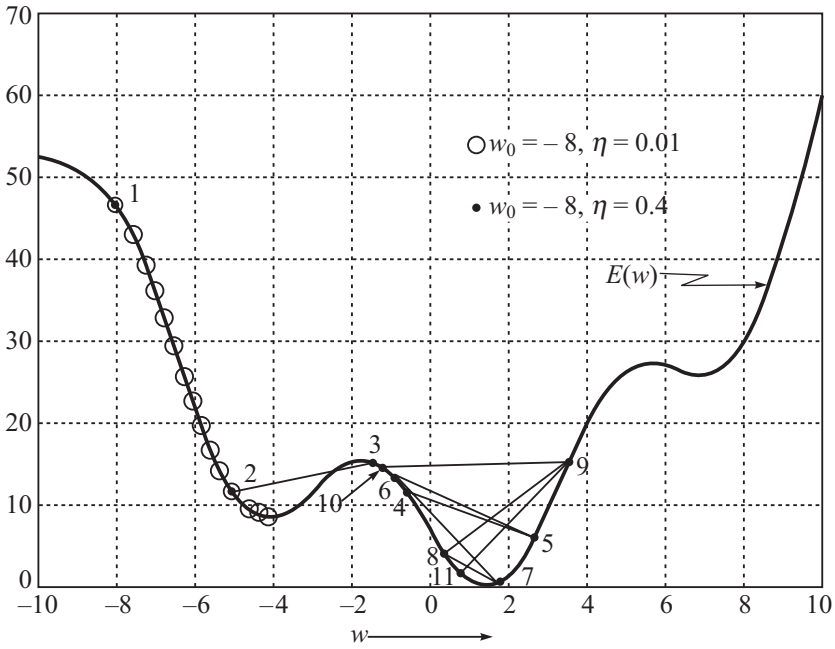


Figure 5.17 Convergence of gradient-descent algorithm

The gradient-descent update law for parameter w is obtained as,

$$\Delta w = -\eta \frac{\partial E}{\partial w} = -\eta (w - 8 \cos w)$$

Figure 5.17 shows the convergence of cost function $E(w)$ with respect to the weight w. It can be seen from the figure that when $\eta = 0.01$ and $w_0 = -8$, the final weight $w = -4$ corresponds to a local minimum (descent indicated by '○' in the figure). When $\eta = 0.4$ and $w_0 = -8$, the local minimum can be avoided and the final weight ultimately settles down at the global minimum (descent indicated by '•' in the figure). It shows that for a small step size, error backpropagation algorithm may get stuck at a local minimum. For a larger step size, it may come out of local minima and get into global minimum. (However, the algorithm may not converge for a large step size). Moreover, when the step size is large, it zigzags its way about the true direction to the global minimum, thus leading to a slow convergence. There is no comprehensive method to select the initial weight w_0 and learning rate η for a given problem so that the global minimum can be reached. When trapped in a local minima, the whole learning procedure must be repeated starting from some other initial weights vector; a costly exercise in terms of computing time.

Despite the lack of assured convergence to the global minimum, error-backpropagation algorithm is a highly effective function approximation method is practice. In many practical applications, the problem of local minima has not been found to be as severe as one might fear. Intuitively, more weights in the network correspond to high dimensional error surfaces (one dimension per weight) that might provide 'escape routes' for gradient descent to fall away from the local minimum.

Despite these comments, gradient descent over the complex error surfaces represented by MLP networks is still poorly understood, and no methods are known to predict with certainty when local minima will cause difficulties. Common heuristics to attempt to alleviate the problem of local minima include:

- Add a momentum term to the weight-update rule (described in the next sub-section).
- Use incremental training rather than batch training. Incremental training effectively descends a different error surface for each training example, relying on the average of these to approximate the gradient with respect to the full training set.

5.7.3 Adding Momentum to Gradient Descent

Despite the fact that error-backpropagation algorithm triggered a revival of the whole neural networks field, it was clear from the beginning that the standard error-backpropagation algorithm is not a serious candidate for finding the optimal weights (the global minimum of the cost function) for large-scale nonlinear problems. Many improved algorithms have been proposed in order to find a reliable and fast strategy for optimizing the learning rate in a reasonable amount of computing time. One of the first, simple yet powerful, improvements of the standard error-backpropagation algorithm is given here—the *momentum gradient descent*.

A general weight-update equation for gradient descent algorithms is given by,

$$w_j(k+1) = w_j(k) - \eta \frac{\partial E(k)}{\partial w_j(k)} \tag{5.57a}$$

Weights w_j are updated at each iteration number k so that the prescribed cost function $E(k)$ decreases; η is the learning rate with $0 < \eta < 1$.

In momentum gradient descent rule, the weight-update equation is altered by making the update on k^{th} iteration depend partially on the update that occurred during the $(k-1)^{\text{th}}$ iteration:

$$w_j(k+1) = w_j(k) - \eta \frac{\partial E(k)}{\partial w_j(k)} + \alpha[w_j(k) - w_j(k-1)] \tag{5.57b}$$

Here, $0 \leq \alpha < 1$ is a constant called the *momentum rate*.

Momentum term $\alpha[w_j(k) - w_j(k-1)]$ allows a network to respond, not only to the local gradient, but also to recent trends in error surface. Without momentum, a network may get stuck in a shallow local minimum; adding momentum may help the NN 'ride through' local minima. Also, with momentum term, the flat regions in the error surface can be crossed at a much faster rate.

The choice of both the learning rate η and momentum rate α is highly problem-dependent and usually a trial-and-error procedure. There have been many proposals on how to improve learning by calculating and using *adaptive learning rate* and *adaptive momentum rate*, which follow and adjust to changes in the nonlinear error surface.

5.7.4 Heuristic Aspects of the Error-backpropagation Algorithm

Multi-Layer Perceptron (MLP) networks are of great interest because they have a sound theoretical basis, meaning that they have *universal function approximation* property with atleast two layers (one layer NNs do not generally have universal function approximation property). According to the basic universal approximation result [86], it is possible for any smooth function $\mathbf{f}(\mathbf{x})$; $\mathbf{x} = [x_1\ x_2\ \ldots\ x_n]^T$, $\mathbf{f}(\cdot) = [f_1(\cdot)\ f_2(\cdot)\ \ldots\ f_q(\cdot)]^T$, to be approximated randomly closely on a compact set in n-dimensional state space for certain (large enough) number m of hidden-layer neurons. In this result, the activation functions are not required on the NN output layer (i.e., the output layer activation functions may be linear). Also, the bias terms on the output layer are not required, even though the hidden layer bias terms are needed.

Despite this sound theoretical foundation concerning the representational capabilities of NNs, and the success of error-backpropagation learning algorithm, there are many practical problems. The most troublesome is the usually long training process, which does not ensure that the best performance of the network (the absolute minimum of the cost function) will be achieved. The algorithm may get stuck at some local minimum, and such a termination with a suboptimal solution will require repetition of the whole training process by changing the structure or some of the learning parameters that influence the iteration scheme.

Further note that, though the universal function approximation property says 'there exists an NN that approximates $\mathbf{f}(\mathbf{x})$', it does not show how to determine such a network. Many practical questions still remain open, and for a broad range of applications, the design of neural networks, their learning procedures and the corresponding training parameters is still an empirical art.

The discussion that follows in this section regarding the structure of a network and learning parameters does not yield conclusive answers, but it does represent a useful aggregate of experience of extensive application of error-backpropagation algorithm and many related learning techniques [3].

Number of Hidden Layers

Theoretical results and many simulations in different fields support the result that there is no need to have more than one hidden layer. But, at times, the performance of the network can be enhanced with the help of a more complex architecture, particularly, when the mapping to be learned is highly complicated—there is a likelihood of the performance being made better by placing more hidden layers with their own weights, after the first layer, with sigmoid hidden units. When the number of hidden layer units in a single hidden-layer network is too large, it may be practical to employ multiple hidden layers, choosing 'long and narrow' networks over 'short and flat' networks. Experimental evidence tends to show that using a two hidden-layer network for continuous functions has sometimes advantages over a one hidden-layer network, as the former requires shorter training times.

A rule of thumb that might be useful is to try solving the problem at hand using an NN with one hidden layer first.

Type of Activation Functions in a Hidden Layer

It has been shown that a two-layer network is capable of solving any nonlinear function approximation problem. This theoretical result does not require the use of sigmoidal nonlinearity. The proof assumes only that nonlinearity is a continuous, smooth, monotonically increasing function that is bounded above and below. Thus, numerous alternatives to sigmoid could be used without a biological justification.

The most serious competitor to MLP networks are the networks that use radial basis functions (RBFs) in hidden layer neurons. The most representative and popular RBF is a Guassian function. Whether a sigmoidal or Gaussian activation function is preferable is difficult to say. Both types have certain advantages and disadvantages, and final choice depends mostly on the problem (data set).

A fundamental difference between these two types of NNs is that feedforward MLP NNs are representative of *global approximation* schemes, whereas NNs with RBFs (typically the Gaussian activation functions) are representative of *local approximation* schemes. The adjectives *global* and *local* are connected with the region of input space of the network for which the NN has nonzero output (RBF networks will be discussed in a later section).

Is it possible to use a mix of different types of activation functions in a layer of a network? The answer is 'theoretically yes'. It is possible to improve the accuracy with the help of a more sophisticated network architecture with mixed activation functions in a layer. When the mapping to be learned is highly complicated, there is a likelihood that the performance may be improved. But, because the implementation and training of the network becomes more complex, it is the practice to apply only one hidden layer of sigmoidal activation functions and an output layer of linear/sigmoidal units.

Number of Neurons in a Hidden Layer

The number of neurons in a hidden layer is the most important design parameter with respect to approximation capabilities of a neural network. Recall that both the number of input components (attributes/features) and the number of output neurons is, in general, determined by the nature of

the problem. Thus, the real representational power of an NN and its generalization capability are primarily determined by the number of hidden-layer neurons. In the case of general nonlinear regression, the main task is to model the underlying function between the given inputs and outputs by filtering out the disturbances embedded in the noisy training dataset. Similar statements can be made for pattern recognition (classification) problems. By changing the number of hidden-layer nodes, two extreme solutions should be avoided: filtering out the underlying function (not enough hidden-layer neurons) and modeling of noise or overfitting the data (too many hidden-layer neurons). These problems have been discussed earlier in Sections 2.4, and 2.2.

The guidelines to select the appropriate number of hidden neurons are rather empirical at the moment. To avoid large number of neurons and the corresponding inhibitively large training times, the smaller number of hidden layer units are used in the first trial. One increases accuracy by adding in steps more hidden neurons.

Variants of Gradient-Descent Procedure

Two main issues in selecting a training algorithm are fast convergence and accuracy of function approximation. The widely used technique to train a feedforward network has been the backpropagation algorithm with gradient descent optimization. The primary drawbacks of this algorithm are its slow rate of convergence, and its inability to ensure global convergence. Several variants have been proposed to improve the speed of response, and also to account for generalization ability and avoidance of local minimum. Standard optimization techniques that use quasi-Newton techniques have been suggested. The issues with quasi-Newton techniques are that the storage and memory need increase as the square of the size of the network. The nonlinear optimization techniques such as Newton-Raphson method or the conjugate gradient method may be adopted for training the feedforward networks. Though these algorithms converge in fewer iterations than the backpropagation algorithm with gradient descent, too much computations per pattern are required. These variants may converge faster in some cases and slower in others. Other algorithms for faster convergence include extended Kalman filtering, and Levenberg-Marquardt. However, all these variants do not come closer to backpropagation algorithm with gradient descent as far as simplicity and ease of implementation is concerned.

The most popular variant used in backpropagation algorithm is probably altering the weight-update rule in the algorithm by ensuring that at least partly the weight update on the k^{th} iteration relies on the update that took place during the $(k-1)^{\text{th}}$ iteration. This augmented learning process has been presented in the earlier sub-section. To handle the overfitting issue for backpropagation learning, the weight decay approach may be used. That is, bring down each weight by a certain small factor during each iteration. This can be achieved by altering the definition of E so that it includes a penalty term which corresponds to the total magnitude of the network weights. The approach keeps weight values small to bias learning against complex decision surfaces/functions (Eqn (2.22)).

Genetic Algorithm (Appendix A) based optimization in neural networks is capable of global search and is not easily fooled by local minima. Genetic algorithms do not use the derivative for the fitness function. Therefore, they are possibly the best tool when the activation functions are not differentiable. Note that genetic algorithm can take an unrealistically huge processing time to find the best solution. Details are given in Section 5.10 on Genetic-Neural systems.

Learning Rate

In the weight-update rule of gradient descent procedures, the term η, called the *learning rate*, appears. It determines the magnitude of the change $\Delta\bar{\mathbf{w}}$ but not its direction. The learning rate controls the stability and rate of adaptation. The general approach to choosing η is to decrease the learning rate as soon as it is observed that adaptation does not converge. The smaller η is, the smoother the convergence of search but the higher the number of iteration steps needed. Descending by small η will lead to the nearest minimum when the error $E(\bar{\mathbf{w}})$ is a nonlinear function of the weights. If $E_{\min}(\bar{\mathbf{w}})$ is larger than the predefined maximally allowed error, the whole learning procedure must be repeated starting from some other initial weights vector. Therefore, working with small η may be rather costly in terms of computing time.

A typical rule of thumb is to start with some larger learning rate and reduce it during optimization. Clearly, what is considered a small or large learning rate is highly problem-dependent, and proper η should be established in the first few runs for a given problem.

In a recent work [87], two novel algorithms have been proposed on *adaptive learning rate* using the Lyapunov stability theory. It is observed that this adaptive learning rate increases the speed of convergence.

Weights Initialization

The learning procedure using the error-backpropagation algorithm begins with some initial set of weights which is usually randomly chosen. However, the initialization is a controlled process. This first step in choosing weights is important because with 'bad' initial choice, the training may get lost forever without any significant learning, or it may stop soon at some local minima (Note that the problems with local minima are not related only to initialization). Initialization by using random numbers is very important in avoiding the effects of symmetry in the network. In other words, all the hidden-layer neurons should start with guaranteed different weights.

A first guess, and a good one, is to start learning with small initial weights. How small the weights must be depends on the training dataset (very small hidden-layer weights must also be avoided). Learning is often a kind of empirical art, and there are many rules of thumb about how small the weights should be. One is that the practical range of hidden-layer initial weights should be $[-2/n, 2/n]$. Another similar rule of thumb is that the weights should be uniformly distributed inside the range $[-2.4/n, 2.4/n]$ (MATLAB uses the techniques proposed by Nguyan and Widrow).

The initialization of output-layer weights should not result in small weights. If the output-layer weights are small, then so is the contribution of hidden-layer neurons to the output error, and consequently the effect of the hidden-layer weights is not visible enough. Also recall that the output-layer weights are used in calculating the error signal terms for hidden-layer neurons. If the output-layer weights are too small, these terms also become very small, which in turn leads to small initial changes in hidden-layer weights; learning in the initial phase will, therefore, be too slow.

Stopping Criterion at Training

The error-backpropagation algorithm resulted from a combination of sum-of-error-squares (Eqn (5.21)) as a cost function to be optimized and the gradient descent method for weights adaptation.

If the training patterns are colored by noise, the minimization of sum-of-error-squares criterion by incremental learning is equivalent to LMS criterion (Eqn (5.27)). The learning process is controlled by the prescribed maximally allowed or desired error E_{des}. More precisely, one should have an expectation about the magnitude of the error at the end of the learning process.

After a learning epoch (the sweep through all the training pairs) is completed in incremental learning process, the sum of error-squares over all the training pairs is accumulated and the total error E_N is compared with the acceptable (desired) value E_{des}. Learning is terminated if $E_N < E_{\text{des}}$; otherwise a new learning epoch is started. In the batch mode, weight updating is performed after the presentation of all the training examples that constitute an epoch. The error E_N is compared with E_{des} after each iteration of a learning epoch.

The sum-of-error-squares is not good as *stopping criterion* because E_N increases with the increase of number of data pairs. The more data, the larger is E_N. Scaling of error function gives a better stopping criterion. The *Root Mean Square Error* (RMSE) is widely used scalar function (Eqn (2.26)):

$$E_{RMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(e^{(i)})^2} \tag{5.58}$$

There will be no need to change the learning algorithm derived earlier. Training is performed using sum-of-error-squares as the cost function (performance criterion), and RMSE is used as a stopping criterion at training. However, if desired, for the batch mode, the learning algorithm with *Mean Square Error* (MSE) (refer to Eqn (2.25))

$$E_{av} = \frac{1}{N}\sum_{i=1}^{N}(e^{(i)})^2 \tag{5.59}$$

may be used as the cost function for training the network.

The condition for termination on the basis of $E_N < E_{\text{des}}$ is, in fact, a weak strategy as backpropagation is vulnerable to overfitting the training examples at the cost of reducing generalization accuracy over other unseen examples. One of the most successful techniques for overcoming the overfitting issue is to merely provide a validation dataset. The algorithm supervises the error with regard to this validation set, while making use of the training set to drive the gradient descent search (refer to Section 2.6).

5.8 MULTI-CLASS DISCRIMINATION WITH MLP NETWORKS

In Section 5.2, we discussed the *linear classification* problem using regression techniques. We found that a single perceptron can be used to realize binary classification.

Multi-Layer Perceptron (MLP) networks can be used to realize *nonlinear classifier*. Most of the real-world problems require nonlinear classification.

Consider the network of Fig. 5.14 with a single log-sigmoid neuron in the output layer. For a binary classification problem, the network is trained to fit the desired values of $y^{(i)} \in [0, 1]$

representing Classes 1 and 2, respectively. The network output $\hat{y}$ approximates $P(\text{Class } 1|\mathbf{x})$; $P(\text{Class } 2|\mathbf{x}) = 1 - \hat{y}$.

When training results in perfect fitting, then the classification outcome is obvious. Even when target values cannot fit perfectly, we put some thresholds on the network output $\hat{y}$. For [0, 1] target values:

$$\text{Class} = \begin{cases} 1 & \text{if } 0 < \hat{y} \leq 0.5 \\ 2 & \text{if } \hat{y} > 0.5 \end{cases} \tag{5.60}$$

What about an example that is very close to the boundary $\hat{y} = 0.5$? Ambiguous classification may result in near or exact ties.

Instead of '0' and '1' values for classification, we may use values of 0.1 and 0.9. 0 and 1 is avoided because it is not possible for the sigmoidal units to produce these values, considering finite weights. If we try to train the network to fit target values of precisely 0 and 1, gradient descent will make the weights grow without limits. On the other hand, values of 0.1 and 0.9 are attainable with the use of the sigmoid unit with finite weights.

Consider now a multi-class discrimination problem. The inputs to the MLP network are just the values of the feature measurements (suitably normalized). We may use a single log-sigmoid node for the output. For the M-class discrimination problem, the network is trained to fit the desired values of $y^{(i)} = \{0.1, 0.2, \ldots, 0.9\}$ for Classes 1, 2, …, 9 (for $M = 9$), respectively. We put some thresholds on the network output $\hat{y}$. For $\{0.1, \ldots, 0.9\}$ desired values:

$$\text{Class} = \begin{cases} 1 & \text{if } 0 < \hat{y} \leq 0.1 \\ 2 & \text{if } 0.1\, \hat{y} \leq 0.2 \\ \vdots & \\ 9 & \text{if } \quad \hat{y} > 0.8 \end{cases} \tag{5.61}$$

Note that the difference between the desired network output for various classes is small; ambiguous classification may result in near or exact ties. The approach may work satisfactorily if the classes are well separated. This gets impractical as the number of classes gets large and the boundaries are artificial.

A more realistic approach is *1–of–M encoding*. A separate node is used to represent each possible class and the target vectors consist of 0s everywhere except for the element that corresponds to the correct class. For a four-class ($M = 4$) discrimination problem, the target vector

$$\mathbf{y}^{(i)} = [y_1^{(i)}\, y_2^{(i)}\, y_3^{(i)}\, y_4^{(i)}]^T$$

To encode four possible classes, we use

$$\mathbf{y}^{(i)} = [1\ 0\ 0\ 0]^T \text{ to encode Class 1}$$

$$\mathbf{y}^{(i)} = [0\ 1\ 0\ 0]^T \text{ to encode Class 2}$$

$$\mathbf{y}^{(i)} = [0\ 0\ 1\ 0]^T \text{ to encode Class 3}$$

and

$$\mathbf{y}^{(i)} = [0\ 0\ 0\ 1]^T \text{ to encode Class 4}$$

We are therefore using four log-sigmoid nodes in the output layer of the MLP network (Fig. 5.15), and binary output values (we want each output to be 0 or 1; instead of 0 and 1 values, we use values of 0.1 and 0.9).

Once the network has been trained, performing the classification is easy: simply choose the element $\hat{y}_k$ of the output vector that is the largest element of $\mathbf{y}$, i.e., pick the $\hat{y}_k$ for which $\hat{y}_l > \hat{y}_q \forall\, q \neq k;\ q = 1, \ldots, M$.

This generates an unambiguous decision, since it is very unlikely that two output neurons will have identical largest output values.

The 1-of-M encoding approach is, in fact, equivalent to designing M binary classifiers.

Neural networks perform best when the features and response variables are on a scale of [0, 1]. For this reason, all variables should be scaled to a $\{0, 1\}$ interval before feeding them into the network. For a numerical variable x that takes values in the range $\{x_{\min}, x_{\max}\}$, we normalize the measurements as follows:

$$x_{\text{norm}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

Even if new data exceed this range by a small amount, this will not affect the results much.

For more details on multiclass classification using neural networks, refer to [59, 88–90].

Example 5.2

Consider the toy dataset shown in Table 5.5, for a certain processed cheese [20]. The two attributes x_1 and x_2 are scores for fat and salt, indicating the relative presence of fat and salt in the particular cheese sample. The output variable y is the cheese sample's acceptance by a taste-test panel.

Table 5.5 A toy dataset

Sample i	x_1 (Fat score)	x_2 (Salt score)	y (Acceptance)	$\mathbf{y} = [y_1\ y_2]^T$ (Target vector)
1	0.2	0.9	yes	$[0.9\ \ 0.1]^T$
2	0.1	0.1	no	$[0.1\ \ 0.9]^T$
3	0.2	0.4	no	$[0.1\ \ 0.9]^T$
4	0.2	0.5	no	$[0.1\ \ 0.9]^T$
5	0.4	0.5	yes	$[0.9\ \ 0.1]^T$
6	0.3	0.8	yes	$[0.9\ \ 0.1]^T$

Figure 5.18 describes an example of a typical neural net that could be used for predicting the acceptance for the given scores on fat and salt. 1-of-M encoding scheme has been used for classification. Two separate nodes have been used to represent two possible classes: 'yes' and 'no'.

The target vector $\mathbf{y} = [0.9\ 0.1]^T$ encodes Class 'yes'; and $\mathbf{y} = [0.1\ 0.9]^T$ encodes Class 'no'. The encoded desired output for each sample is shown in Table 5.5.

We choose the following initial weight and bias values: $w_{11} = 0.05$, $w_{21} = -0.01$, $w_{31} = 0.02$, $w_{12} = 0.01$, $w_{22} = 0.03$, $w_{32} = -0.01$, $w_{10} = -0.3$, $w_{20} = 0.2$, $w_{30} = 0.05$, $v_{11} = 0.01$, $v_{21} = -0.01$, $v_{12} = 0.05$, $v_{22} = 0.03$, $v_{13} = 0.015$, $v_{23} = 0.02$, $v_{10} = -0.015$, $v_{20} = 0.05$.

We use a learning rate $\eta = 0.5$ to update weight and bias values. The activation function for hidden and output nodes is unipolar sigmoid.

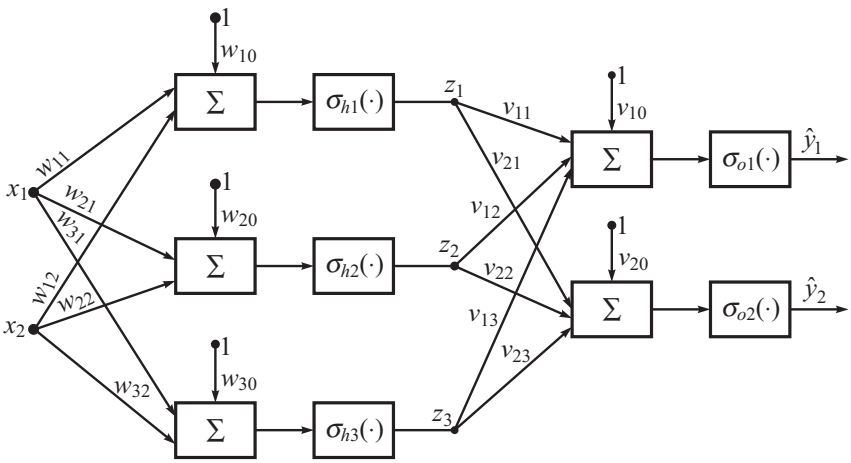


Figure 5.18 A typical neural net for Example 5.2

Sample calculations may be carried out in a way similar to that used in Example 5.1. The reader is encouraged to do this practice on the first sample of the training data.

5.9 RADIAL BASIS FUNCTIONS (RBF) NETWORKS

Let us consider our feature vectors $\mathbf{x}$ to be in the n-dimensional space $\Re^n$. Let $\phi_1(\cdot),\phi_2(\cdot),\ldots,\phi_m(\cdot)$ be m nonlinear functions

$$\phi_l: \Re^n \to \Re;\ l = 1, \ldots, m \tag{5.62a}$$

which define the mapping

$$\mathbf{x} \in \Re^n \to \mathbf{z} \in \Re^m \tag{5.62b}$$

$$\mathbf{z} = \begin{bmatrix} \phi_1(\mathbf{x}) \\ \phi_2(\mathbf{x}) \\ \vdots \\ \phi_m(\mathbf{x}) \end{bmatrix}$$

Our goal now is to investigate whether there is an appropriate value for m and functions $\phi_l(\cdot)$ so that a nonlinear function $f(\mathbf{x})$ can be approximated as a linear combination of $\phi_l(\mathbf{x})$, that is,