Chapter 1
Introduction to Neural Networks

1.1 Introduction

Parallel computation and neural networks are new computing paradigms that are finding increasing attention among computer and artificial intelligence scientists. The key element of these paradigms is a novel computational structure composed of a large number of highly interconnected processing elements (neurons) working in parallel. Therefore, many operations can be performed simultaneously, as opposed to traditional serial processing in which computations must be performed in sequential order. Simple neural networks were built in the 1950s, but little progress was made in the field due, both to the lack of proper technology and to the breakthroughs in other areas of artificial intelligence. The increasing power of available computers in the 1970s and the development of efficient parallel computing techniques renewed the interest in this field. Nowadays, neural networks have already proven successful for solving hard problems which at first seem to be intractable and difficult to formulate using conventional computing techniques. Examples of such problems can be found in a great variety of domains such as pattern recognition (see Ripley (1996) and Bishop (1997), and the references therein), vision and speech recognition (see Allen (1995) and Skrzypek and Karplus (1992)), time series prediction and forecasting (see Azoff (1994) and Myers (1992)), process control (Miller et al. (1995)), signal processing (Cichocki et al. (1993)), etc.

The aim of this chapter consists of introducing the fundamental concepts of neural computation as well as describing some of the most popular neural network architectures: the Hopfield, the feed-forward, and the competitive neural networks. It is not our aim to give an up-to-date detailed description of this broad field; for more details we refer the reader to some books devoted entirely to this discipline. For instance, Freeman and Skapura (1991) and Rumelhart and McClelland (1986) keep the mathematical treatment to a minimum; Hertz, Krog, and Palmer (1991) provide a rigorous analysis and present a statistical mechanics point of view; Lisboa (1992) and Swingler (1996) describe some real world applications of this field, and Anderson and Rosenberg (1988) contains reprinted versions of the classical introductory papers of this field. The interested reader may also consult some interesting WWW sites containing tutorials[1] and software[2] related to neural networks.

We start with some basic ideas that inspired parallel computing in Section 1.2. Section 1.3 discusses the general structure of neural networks and their main components: the network architecture and the neuron functions. The problems of learning from data and validating the resulting model are analyzed in Section 1.4. Section 1.5 deals with the validation problem. Section 1.6 describes one of the most popular architectures, the Hopfield network, giving an example of application to character recognition. Then, perceptrons and multi-layer perceptrons are presented in Sections 1.7 and 1.8, respectively. Examples of classification, function approximation, time series prediction, regression models, and modeling chaotic dynamics are also presented to illustrate the concepts and the wide domain of applications of these architectures. In Section 1.9 some examples of feed forward neural network are given. Finally, a brief introduction to competitive architectures is presented in Section 1.10.

1.2 Inspiration from Neuroscience

Before the appearance of neural networks and parallel computing, all the standard computing methods and tools used for information processing had some common characteristics:

- the *knowledge was explicitly represented* using rules, semantic nets, probabilistic models, etc.,

[1]A growing list of online Web tutorials can be found in the PNNL homepage *http://www.emsl.pnl.gov:2080/proj/neuron/neural/what.html*. An interesting example with several references is the neural nets online tutorial by Kevin Gurney *http://www.shef.ac.uk/psychology/gurney/notes/index.html*

[2]Updated lists of NN software are given, for example, in the NeuroNet network's site *http://www.neuronet.ph.kcl.ac.uk/neuronet/* and in the PNNL site *http://www.emsl.pnl.gov:2080/proj/neuron/neural/systems/*.

- *the human logical reasoning process was imitated* for problem solving, focusing on actions and underlying motives (rule chaining, probabilistic inference), and
- *the information was sequentially processed.*

The quick development of some artificial intelligence fields such as pattern recognition during the last two decades uncovered a great number of hard problems where no explicit representation of knowledge was suitable and no logical reasoning process was available. Therefore, standard algorithmic approaches and computational structures were inappropriate to solve these problems. Artificial neural networks were introduced as alternative computational structures, created with the aim of reproducing the functions of the human brain. The brain is composed of about 10^{11} neurons which receive electrochemical signals from other neurons through the synaptic junctions which connect the axon of the emitting and the dentrites of the receiving neurons (the axon of a typical neuron makes a few thousand synapses with other neurons). Based on the received inputs, the neuron computes and sends its own signal. The emission process is controlled by the internal potential associated with a neuron. If this potential reaches a threshold, an electrical pulse is sent down the axon; otherwise, no signal is sent.

The computational models known as neural networks are inspired from the above neurophysiological characteristics and, therefore, are formed by a large number of weighted connections among several layers of processors, or neurons, which perform simple computations. Neural networks do not follow rigidly programmed rules, as more conventional digital computers do. Rather, they use a learn-by-analogy learning process, i.e., the connection weights are automatically adjusted to reproduce a representative set of training patterns with the aim of capturing the structure of the problem. This is also inspired in the way learning occurs in neurons, changing the effectiveness of the synapses, so that the influence of one neuron on another changes. It is important to remark here that current network architectures and neuron functions are extremely simplified when seen from a neurophysiological point of view, though they provide new computing structures to solve many interesting problems.

1.3 Components of Neural Networks

The following definitions describe the main components of an artificial neural network (ANN).

Definition 1.1 (Neuron or Processing Unit) *A neuron, or processing unit, over the set of nodes N, is a triplet (X, f, Y), where X is a subset of N, Y is a single node of N and $f : \mathbb{R} \to \mathbb{R}$ is a neural function (also called* *threshold or activation function) which computes an output value for Y based on a linear combination of the nodes X, i.e.,*

$$Y = f(\sum_{x_i \in X} w_i\, x_i).$$

The elements X, Y and f are called the set of input nodes, the set of output nodes, and the neuron function of the neuron unit, respectively.

Definition 1.2 (Artificial Neural Network) *An artificial neural network (ANN) is a pair (N, U), where N is a set of nodes and U is a set of processing units over N, which satisfies the following condition: Every node $X_i \in N$ must be either an input or an output node of at least one processing unit in U.*

Figure 1.1(a) shows an example of a neural network with eight nodes $\{x_1, \ldots, x_8\}$ containing five processing units:

$$\begin{aligned} U_1 &= (\{x_1, x_2, x_3\}, f_1, \{x_4\}), \\ U_2 &= (\{x_1, x_2, x_3\}, f_2, \{x_5\}), \\ U_3 &= (\{x_1, x_2, x_3\}, f_3, \{x_6\}), \\ U_4 &= (\{x_4, x_5, x_6\}, f_4, \{x_7\}), and \\ U_5 &= (\{x_4, x_5, x_6\}, f_5, \{x_8\}). \end{aligned} \tag{1.1}$$

Note that, for the sake of simplicity, the neuron functions are not explicitly represented in Figure 1.1(a). Figure 1.1(b) shows a detailed description of a typical neuron unit.

The following sections analyze in detail the components of a neural network.

1.3.1 Neurons: Processing Units

The neurons are the processing elements of the neural network. As we have seen in Definition 1.1, a typical neuron, say $(\{x_1, \ldots, x_n\}, f, y_i)$, performs a simple computation with the inputs to obtain an output value:

$$y_i = f(\sum_{j=1}^{n} w_{ij} x_j), \tag{1.2}$$

where $f(x)$ is the neuron function and the weights w_{ij} can be positive or negative, reproducing the so called excitatory or inhibitory character of neuron synapses, respectively. Sometimes we use the output node y_i to refer to the whole neuron unit.

Of importance in neural networks is the concept of *linear activity of a neuron y_i*, which is simply the weighted sum of the inputs from other

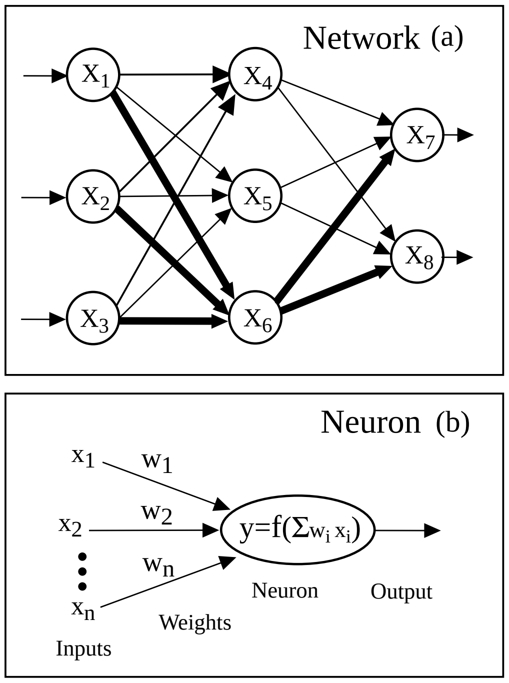


FIGURE 1.1. (a) An artificial neural network and (b) the processing function of a single neuron.

neurons[3]:

$$Y_i = \sum_{j=1}^{n} w_{ij} x_j. \tag{1.3}$$

Therefore, the output of a neuron y_i is simply obtained by transforming the linear activity using the activation function (see Figure 1.2).

In some cases, in order to account for a threshold value θ_i for the neuron y_i, a new auxiliary neuron $x_0 = -1$ can be connected to y_i with a weight $w_{i0} = \theta_i$.

$$Y_i = \sum_{j=1}^{n} w_{ij} x_j - \theta_i = \sum_{j=0}^{n} w_{ij} x_j. \tag{1.4}$$

Figure 1.2 illustrates the computations involved in the neuron processing. This definition of neuron activity was first suggested by McCulloch and Pitts (1943) as a simple mathematical model of neural activity. In particular, they considered a binary step activation function.

[3]With the aim of keeping the notation as simple as possible we shall use uppercase letters, Y_i, to refer to the linear activity of the neuron, before being processed by the neuron function.

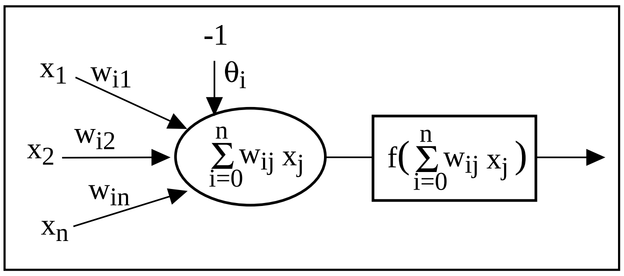


FIGURE 1.2. Schematic neuron computation consisting of a weighted sum of the inputs from other neurons, including a threshold value θ_i, and a processing activity function $f(x)$.

The most popular activation functions are described below.

- **Linear functions:** which give a linear output:

$$f(x) = x;\ x \in \mathbb{R}.$$

- **Step functions:** which give a binary output depending only on the position, below or above a given threshold value. Examples of these functions are the sign, $sgn(x)$, and the standard step, $\Theta(x)$, functions defined as,

$$sgn(x) = \begin{cases} -1, \text{ if } x < 0, \\ 1, \text{ otherwise}, \end{cases}, \quad \Theta(x) = \begin{cases} 0, \text{ if } x < 0, \\ 1, \text{ otherwise}. \end{cases}$$

  Figure 1.3 compares both the linear and the sign activation functions. The linear function gives a gradual response of the input, whereas the nonlinear threshold function determines a firing threshold for the activity (below the threshold there is no activity and above it the activity is constant). In this case, the neuron function results

$$y_i = sgn(Y_i) = sgn(\sum_{j=0}^{n} w_{ij} x_j). \tag{1.5}$$

- **Sigmoidal functions:** Bounded monotonic functions which give a nonlinear gradual output for the inputs. The most popular sigmoidal functions are

  1. The logistic function from 0 to 1 (see Figure 1.4):

$$f_c(x) = \frac{1}{1 + e^{-c\,x}}.$$

  2. The hyperbolic tangent function from -1 to 1 (see Figure 1.5):

$$f_c(x) = tanh(c\,x).$$

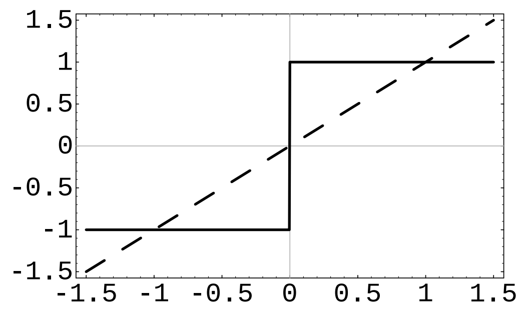


FIGURE 1.3. Linear and sign activation functions.

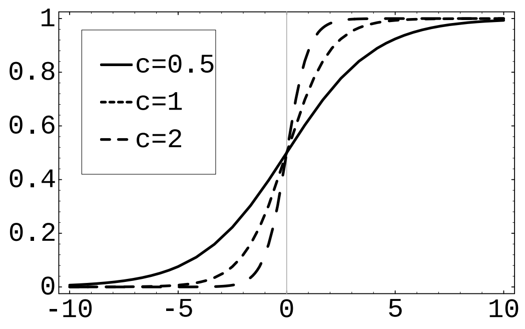


FIGURE 1.4. Sigmoidal logistic activation functions $f_c(x) = (1 + e^{-c\,x})^{-1}$.

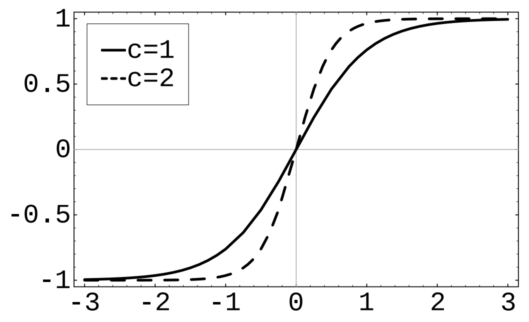


FIGURE 1.5. Hyperbolic tangent activation functions $f_c(x) = tanh(c\,x)$.

1.3.2 Network Architectures

The neurons can be organized in layers connected by several types of links including *forward*, *lateral*, and *delayed* connections:

1. *Forward connections*: They connect neurons on a given layer with neurons in the next layer of the network (see Figure 1.6(a)). Each of these connections imply a functional composition between the acti-

vation functions of the corresponding neurons and, as we shall see, gives the neural network the capability of approximating a wide class of nonlinear phenomena.

2. *Lateral connections*: They connect neurons within the same layer. Besides the simple case of single-layer ANNs (see Figure 1.6(b)), this type of connection is mostly used in competitive layers, where each node is connected to itself via a positive (excitatory) weight and to all other nodes in the layer with negative (inhibitory) weights (see Figure 1.6(c)).

3. *Delayed, or recurrent, connections:* Incorporated to the network to deal with dynamic and temporal models, i.e., models with memory.

The network architecture can be represented by a weight matrix $W = (w_1, \ldots, w_n)$, where w_i is the vector containing the weights of the connections from other neurons, say x_j, to neuron x_i: $w_i = (w_{i1}, \ldots, w_{ij}, \ldots, w_{in})$.

In some cases, the topology of the network allow us classifying the neuron units in a natural way as follows:

Definition 1.3 (Input Neuron of a Neural Network) *A neuron is said to be an input neuron of a neural network* (X, U)*, if it is the input of at least one functional unit in* U *and is not the output of any processing unit in* U*.*

Definition 1.4 (Output Neuron of a Neural Network) *A neuron is said to be an output neuron of a functional network* (X, U)*, if it is the output of at least one functional unit in* U *and is not the input of any processing unit in* U*.*

Definition 1.5 (Hidden or Intermediate Neuron of a Neural Network) *A neuron is said to be an intermediate neuron of a neural network* (X, U)*, if it is the input of at least one functional unit in* U *and, at the same time, is the output of at least one processing unit in* U*.*

One of the most popular and powerful network architectures is the so called *feed forward network*, or multi-layer perceptron, which is formed by an input layer, an arbitrary number of hidden layers, and an output layer. Each of the hidden and output neurons receive an input from the neurons on the previous layer (forward connections). Figure 1.6(a) shows a feed forward neural network with four input neurons, three hidden units in a single layer, and two output neurons.

Another popular and simple architecture is the *Hopfield network* which contains a single layer and all the possible lateral connections between different neurons (see Figure 1.6(b)). In this case, as we shall see in Section 1.6, all the neurons play the role of inputs, hidden and output neurons.

An ANN can also include competitive layers where the neurons compete to gain the largest activity for a given pattern (see Figure 1.6(c)).

These types of network structures are analyzed in detail in Sections 1.6, 1.7, and 1.10.

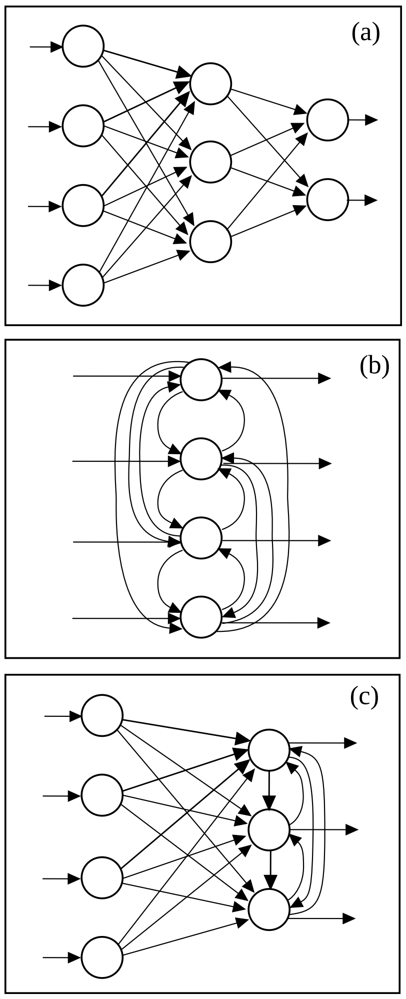


FIGURE 1.6. (a) Multi-layer feed forward network with four input, three hidden and two output neurons. (b) Four nodes Hopfield network. (c) Two-layer feed forward network with competitive output layer.

1.4 Learning

One of the main properties of ANNs is their ability to learn from data. Once the network architecture for a particular problem has been chosen, the weights of the connections have to be adjusted to encode the information contained in a set of *training data* in the network structure. The different learning methods can be classified in two categories, depending on the type of available information:

- *Supervised learning:* The training patterns, $\{(a_p; b_p), p = 1, \ldots, r\}$, are instances of a vector of input variables, a, together with the associated outputs b. Therefore, each output neuron is told the desired response to input signals. In this case, the weights are usually obtained by minimizing some error function which measures the difference between the desired output values and those computed by the neural network. An important issue of this type of learning is the problem of error convergence. In general, the resulting error function may contain multiple local minima where the convergence to the optimal global minima may not be obtained.

- *Unsupervised learning:* In this case, the data is presented to the network without any external information and the network must discover by itself patterns, or categories. This type of learning is also referred to as self-organization. Some special unsupervised learning methods are:

  - *Hebbian learning*, which consists of modifying the weights according to some correlation criteria among the neuron activities (see Hebb (1949)),
  - *Competitive learning*, where different neurons are connected with negative (inhibitory) weights which force competition to gain neuron activity, and
  - *Feature mapping*, which is concerned with the geometric arrangement of the weight vectors of competitive units (see Kohonen (1997)). In unsupervised learning there may be both input and output neurons but, as opposed to supervised learning, there is no information about what output corresponds to each of the inputs in the training data.

1.5 Validation

Once the learning process has finished and the weights of the neural network have been calculated, it is important to check the quality of the resulting model. For example, in the case of supervised learning, a measure of the quality can be given in terms of the errors between the desired and the computed output values for the training data. Some standard error measures are:

1. The Sum of Square Errors (SSE), defined as
$$\sum_{p=1}^{r} \parallel b_p - \hat{b}_p \parallel^2 . \qquad (1.6)$$

2. The Root Mean Square Error (RMSE) defined as
$$\sqrt{\sum_{p=1}^{r} \parallel b_p - \hat{b}_p \parallel^2 /r}. \qquad (1.7)$$

3. The Maximum Error,
$$max\{\parallel b_p - \hat{b}_p \parallel,\ p = 1, \ldots, r\}, \qquad (1.8)$$

   where $\hat{b}_p$ is the network output for the input vector a_p. Note that in the case of a single output, the norm function $\parallel . \parallel$ reduces to the usual absolute value function $| . |$.

It is also desirable to perform a *cross-validation* to obtain a measure of the prediction quality of the model. To this aim, the available data can be divided in two parts: one part for training and the other for testing. When the test error is much larger than the training error, then an *over-fitting problem* has occurred during the training process. It is a well known result in statistics that when using a model with many parameters to fit a set of data with only a small degrees of freedom, then the obtained model may not capture the real trends of the underlying process, even though it can present a small training error. In this case, the training process reduces to a interpolation of the training data, including the noise, with a complicate sigmoidal function.

The over-fitting problem is illustrated with the following example. The curve shown in Figure 1.7(a) passes exactly through the training points (represented by dots). It is, in fact, the seven-degree interpolating polynomial of the eight data points, so it is a zero-error model. However, if we consider an alternative set of test data from the underlying model, then the above curve may not predict accurately the new data, as it is the case in Figure 1.7(b). The problem is the excessive number of parameters (the eight polynomial coefficients) contained in the model. In this case, a more realistic model could be obtained by considering a smaller number of parameters (the third degree polynomial shown with dashed lines in Figure 1.7(c)). Now, the approximation error is similar both for the training and test data sets, indicating that we have captured the actual trends of the model. The difference between both models can be seen in Figure 1.7(d).

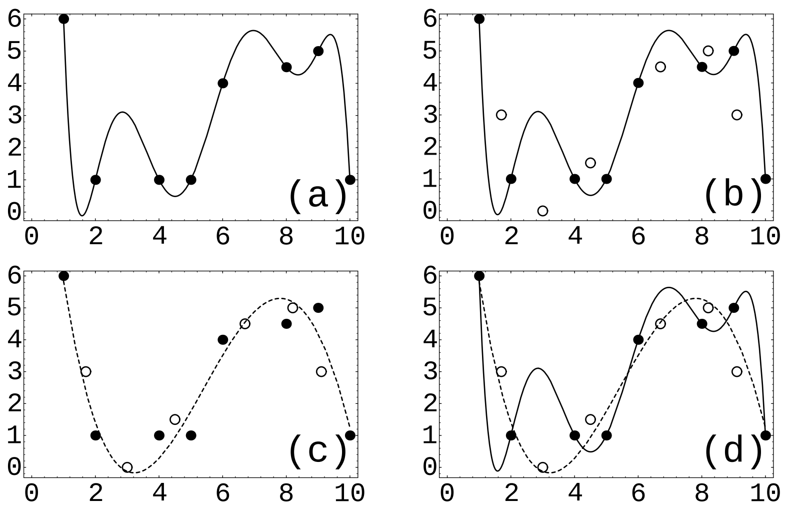


FIGURE 1.7. Illustration of the overfitting problem. (a) A model with too many parameters is used to fit the training data. (b) A large error is obtained on a test data set. (c) A model with less parameters approximate both the training and test data. (d) Both models are compared.

1.6 The Hopfield Neural Network

The Hopfield neural network is a one-layer architecture which is mainly used as an autoassociative memory to store and recall information. The information is stored by using a simple unsupervised learning method which obtains the corresponding weight matrix. Thereafter, using an arbitrary configuration of the neurons as input, the network will settle on exactly that stored image which is nearest to the starting configuration in terms of Hamming distance[4]. Thus, given an incomplete or corrupted version of the stored information, the network is able to recover the corresponding original image.

Consider a single-layer neural network containing a set of binary neurons $\{x_1, \ldots, x_n\}$ (with values in $\{-1, 1\}$) where each neuron x_i is connected to all others with weights $w_i = (w_{i1}, \ldots, w_{in})$, with $w_{ii} = 0$ indicating that no self-connections exist (see Figure 1.6(b)). Consider also the following binary definition of the neuron, obtained from (1.3) with the sign step activation

[4]The Hamming distance between two binary codes $a_1 \ldots a_n$ and $b_1 \ldots b_n$ is the ratio of the number of different bits $a_j \neq b_j$ divided by the total number of bits n.

function:

$$x_i = sgn(\sum_{j=1}^{n} w_{ij} x_j). \tag{1.9}$$

Now, suppose that we want to obtain the appropriate weights to "memorize" a pattern $a = (a_1, \ldots, a_n)$. Then, the weights must satisfy the following stability conditions:

$$a_i = sgn(\sum_{j=1}^{n} w_{ij} a_j),\ i = 1 \ldots, n, \tag{1.10}$$

so that the network returns the same pattern when given it as input. Since we are using the neuron values $\{-1, 1\}$, then $a_j^2 = 1$ and the above stability conditions can be achieved by considering the weights

$$w_{ij} = \frac{1}{n} a_i a_j. \tag{1.11}$$

This formula is related to the concept of Hebbian learning. The main idea of Hebbian learning (see Hebb (1949)) consists of modifying the weights according to the existing correlation among the connected neurons. Then, when given the pattern a as input, we have

$$x_i = sgn(\frac{1}{n} \sum_{j=1}^{n} a_i a_j\, a_j) = sgn(a_i) = a_i,\ i = 1 \ldots, n.$$

The same algorithm can be extended to several patterns, $\{(a_{p1}, \ldots, a_{pn})$, $p = 1, \ldots, r\}$, in the following way:

$$w_{ij} = \frac{1}{n} \sum_{p=1}^{r} a_{pi} a_{pj}. \tag{1.12}$$

Then, when given a pattern a_p as input we obtain

$$\begin{aligned} x_i &= sgn(\frac{1}{n} \sum_j \sum_{k=1}^{r} a_{ki} a_{kj}\, a_{pj}) \\ &= sgn(\frac{1}{n} \sum_j a_{pi} a_{pj}\, a_{pj} + \sum_j \sum_{k \neq p} a_{ki} a_{kj}\, a_{pj}) \\ &= sgn(a_{pi} + \frac{1}{n} \sum_j \sum_{k \neq p} a_{ki} a_{kj}\, a_{pj}). \end{aligned} \tag{1.13}$$

In this case, the problem of stability is determined by the cross correlation terms in the second term of the sum in (1.13). If it is smaller than n, then we can conclude that the pattern is stable. An statistical analysis of these crossed terms has shown that less than 1% of the bits will be unstable when the number of patterns to be stored, p, and the number of neurons, n, satisfy the relationship: $p < 0.138\,n$ (see McEliece et al. (1987)).

Equation (1.12) gives a simple straightforward unsupervised learning algorithm to store and recall patterns using a Hopfield network. An alternative learning algorithm is based on an analogy of this architecture with some well-known statistical physics phenomena. This analogy is established through an energy function (a function of the weights),

$$E(w) = -\frac{1}{2}\sum_{i,j} w_{ij} x_i x_j, \tag{1.14}$$

to be minimized using a gradient descendent technique (for further information see Hopfield (1982)).

1.6.1 Example: Storing and Retrieving Patterns

In this section we illustrate the performance of Hopfield networks to store and recall a set of bitmap images. More specifically, in this example we consider the problem of recognizing handwritten characters. We recommend the reader to work out this example by using some of the available free programs implementing this type of architecture[5].

We use a network input consisting of a 5×5 grid that receive a pixel image of a particular handwritten digit. In this example, we consider the three standardized representations of the letters 'A', 'E', and 'I' shown in Figure 1.8 and use a 25 neuron Hopfield network to store these patterns. Each of the neurons is associated with one of the pixels, starting from the upper-left corner of the image. Thus, the digits 'A', 'E', and 'I' are represented as '-1-11-1-1...', '1111-1...', and '-1111-1...', respectively where negative values are represented in gray and positive in black. The resulting 25×25 matrix of weight connections is shown in Figure 1.9. This matrix was simply obtained by using (1.12). For instance,

$$w_{12} = \frac{1}{25}\sum_{k=1}^{3} a_{1k} a_{2k} = \frac{1}{25}(-1 \times -1 + 1 \times 1 + 1 \times -1) = \frac{1}{25} \times 1.$$

This value, ignoring the normalizing constant, is represented by the small-size black box in the upper-left corner of Figure 1.9 (weight w_{12}). In this

[5]A demo version of Trajan 3.0 NN Simulator for Win95 can be obtained in *http://www.trajan-software.demon.co.uk/*. Mac users can get the Mactivation package in *http://www.src.doc.ic.ac.uk/public/packages/mac/mactivation/* or by anonymous ftp at the directory */pub/cs/misc/* at *bruno.cs.colorado.edu.* The source code of a C language implementation of this architecture can be obtained at *http://www.geocities.com/CapeCanaveral/1624/hopfield.html.*

figure, the black and gray colors are associate with positive and negative weights, respectively, where the size of the box represents its value.

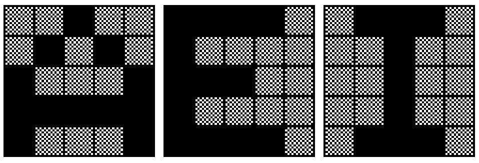


FIGURE 1.8. Training a 5×5 Hopfield network to recognize the vowels 'A', 'E', and 'I'.

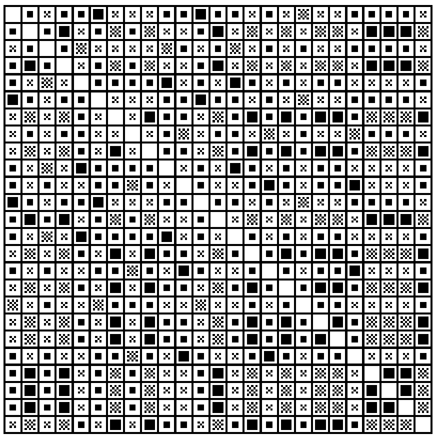


FIGURE 1.9. Matrix of weight connections among the 25 neurons. Positive weight values are represented with black color and negatives with gray. On the other hand, the absolute value of the weight is determined by the box size.

Once the Hopfield network has been trained, it is able to recognize the three vowels when using as input any other 5×5 grid corresponding to a particular non-standardized handwritten version digit. For example, Figure 1.10 shows three input digits (left column), which are corrupted versions of the stored patterns, and the correct patterns returned by the network (right column).

Similarly, if we try to use the same network architecture to recognize the five vowels shown in Figure 1.11, then we obtain the weight matrix shown in Figure 1.12. However, since in this case the number of stored patterns is larger than $0.138 \times n = 0.138 \times 25 = 3.45$, then some stable spurious states can appear in the model, making the whole network useless to recover the stored characters, even from the original information. For example, if we try to recover the digits 'A' and 'I' from the stored images shown on the left in Figure 1.8, then the network returns the spurious states on the right.

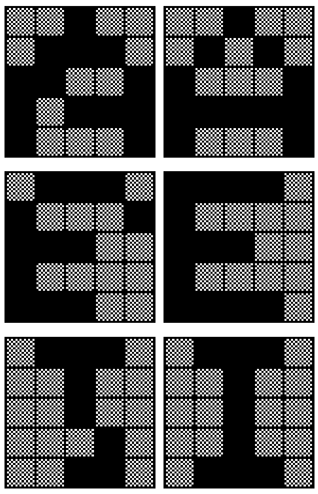


FIGURE 1.10. Recalling stored patterns from corrupted information. The images on the right were recalled by the network after presenting the input images shown on the left.

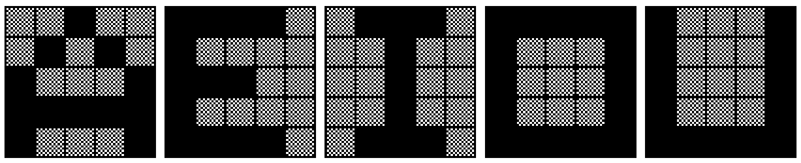


FIGURE 1.11. Training a 5 × 5 Hopfield network to recognize the five vowels.

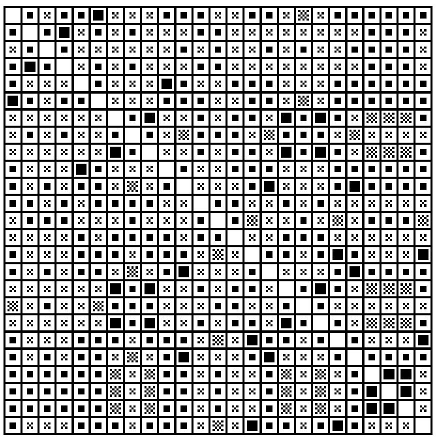


FIGURE 1.12. Matrix of weights connections between the 25 neurons. Positive weight values are represented with black color and negatives with gray.

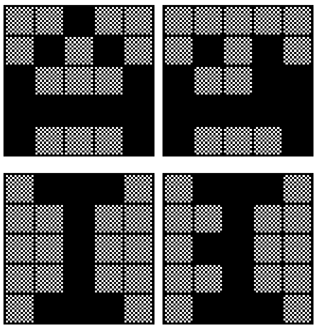


FIGURE 1.13. Stable spurious states appear in the neural network.

1.7 Feed Forward Networks: Perceptrons

In feed forward networks, the neurons are organized in different layers and each of the neurons in one layer can receive an input from units in the previous layer (see Figure 1.6(a)). Perceptrons are the simplest of these network architectures and consist of an input, $\{x_1, \ldots, x_n\}$, and an output, $\{y_1, \ldots, y_m\}$, layers (see Rosenblat (1962)). This type of neural networks are also referred to as $n : m$ perceptrons, indicating the number of inputs and outputs[6]. Figure 1.14 shows a simple $4 : 3$ perceptron.

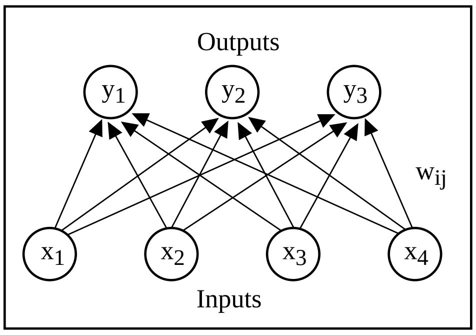


FIGURE 1.14. Single-layer perceptron architecture with 4 inputs and 3 outputs.

A typical output unit y_i performs the computation:

$$y_i = f(Y_i) = f(\sum_{j=1}^{n} w_{ij} x_j), \; i = 1, \ldots, m.$$

[6]Perceptrons are also referred to as single-layer feed forward networks, or single-layer perceptrons. The input layer the is not included in the total number of layers, since it does not perform any computation.

where $f(x)$ is the activation function and w_i the corresponding weight vector. Note that upper case letters indicate the linear activation of a neuron, i.e., the weighted combination of inputs.

The problem of learning this type of neural networks reduces to obtaining the appropriate weights for approximating a given set of input-output patterns $(a_{p1}, \ldots, a_{pn}; b_{p1}, \ldots, b_{pm})$, so that the network outputs $(\hat{b}_{p1}, \ldots, \hat{b}_{pm})$ obtained for the neurons $(y_1, \ldots, y_m)$ when giving the input values $(a_{p1}, \ldots, a_{pn})$ be as close as possible to the desired output values $(b_{p1}, \ldots, b_{pm})$.

1.7.1 Learning Algorithms

In this section we analyze two popular algorithms to train perceptrons. The first of them is inspired on the analogy with neuronal synapses reinforcement proposed by Hebb. The second uses a mathematical optimization method for obtaining the weight configuration which minimizes an error function. As we shall see, in the case of linear perceptrons, both the heuristic and mathematical methods coincide.

- **Hebbian learning:** First, the weights are randomly chosen. Then, the patterns are taken in turn, one at a time, and the weights are modified according to the existing correlation among the input values and the resulting error:

$$\Delta w_{ij} = -\eta(b_{pi} - \hat{b}_{pi})a_{pj}, \tag{1.15}$$

  where the parameter η is the *learning rate*, which indicates the rate of change for the weights. Note that, when $b_{pi} = \hat{b}_{pi}$ (same predicted and desired outputs), then the weight is not modified.

- **Gradient descend method (the delta-rule):** The weights are initially set to random values, as in the Hebbian learning. The idea of this method is using an iterative process which minimizes the sum of squared errors function:

$$E(w) = \frac{1}{2}\sum_{i,p}(b_{pi} - \hat{b}_{pi})^2.$$

  The simplest optimization algorithm consists of moving down the slope of the error function in the space of weight configurations (gradient descendant method). This can be done by iteratively modifying each of the weights w_{ij} by an amount Δw_{ij} proportional to the error gradient. Then, we have:

$$\begin{aligned} \Delta w_{ij} &= -\eta \frac{\partial E(w)}{\partial w_{ij}} = -\eta \sum_p (b_{pi} - \hat{b}_{pi}) \frac{\partial \hat{b}_{pi}}{\partial w_{ij}} a_{pj}, \\ &= -\eta \sum_p (b_{pi} - \hat{b}_{pi}) f'(B_{pi}) a_{pj}, \end{aligned} \tag{1.16}$$

where B_{pi} is the linear activation of the neuron b_{pi}, i.e., $b_{pi} = f(B_{pi})$ and the parameter η gives the learning rate.

In the linear case ($f(x) = x$) the above learning formula reduces to

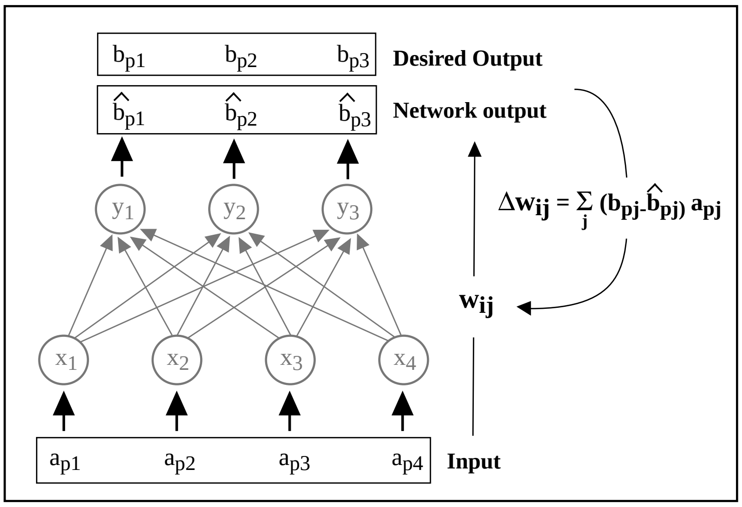
$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = \eta \sum_p (b_{pi} - \hat{b}_{pi}) a_{pj}, \tag{1.17}$$

which is the same as the learning algorithm in (1.15). This simple learning method is schematically illustrated in Figure 1.15. On the other hand, in the case of perceptrons using continuous sigmoidal functions, the formula (1.17) does not involve formal derivatives, since

$$f(x) = \frac{1}{1 + e^{-c\,x}} \quad \Rightarrow \quad f'(x) = c\,f(x)\,(1 - f(x)),$$

and

$$f(x) = tanh(c\,x) \quad \Rightarrow \quad f'(x) = c\,(1 - f(x)^2).$$



b_{p1} b_{p2} b_{p3}
Desired Output
$\hat{b}_{p1}$ $\hat{b}_{p2}$ $\hat{b}_{p3}$
Network output
$\Delta w_{ij} = \sum_j (b_{pj} - \hat{b}_{pj})\, a_{pj}$
y_1 y_2 y_3
w_{ij}
x_1 x_2 x_3 x_4
a_{p1} a_{p2} a_{p3} a_{p4}
Input

FIGURE 1.15. Schematic illustration of the learning algorithm for linear perceptrons. The error between the desired and output values for a given input a_p is used to modify the weights.

1.7.2 Improvements and Modifications

With the aim of improving the efficiency of the above learning methods, several modifications have been proposed in the literature. One of them

x	y	c	x	y	c	x	y	c
0.272	0.987	0	0.524	0.196	1	0.629	0.232	1
0.506	0.371	1	0.750	0.594	1	0.818	0.295	1
0.526	0.900	0	0.005	0.972	0	0.112	0.318	0
0.932	0.968	1	0.641	0.926	0	0.351	0.813	0
0.369	0.938	0	0.827	0.617	1	0.739	0.706	1

TABLE 1.1. Data points (x, y) classified in two categories represented by $c = 0$ and $c = 1$, respectively.

consider an extra momentum term in Δw_{ij} to accelerate the convergence to the minimum. Then, the new updating rule is given by

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} + \alpha \, \Delta' w_{ij},$$

where $\Delta' w_{ij}$ refers to the previous value of Δw_{ij} (in the previous iteration step) and α is the *momentum parameter*.

Other methods include extra terms in the error function to penalize large weights:

$$E(w) = \sum_{p=1}^{r} (y_p - \hat{y}_p)^2 + \lambda \sum_{i,j} w_{ij}^2, \tag{1.18}$$

where λ is a regularization parameter, which controls the balance between fitting the model and avoiding the penalty. The effect of this regularization of the weights is to smooth the error function, since large weights are usually associate with high output values (for a more detailed description of regularization from a statistical point of view see Hoerl (1970)). This technique is also related to the method of *weight decay* which consists of pruning unimportant network connections.

1.7.3 Examples

In this section we present some examples to illustrate the performance of perceptrons. We start by analyzing one of the most typical applications: a linear classification problem. Then, we show how perceptrons generalize standard regressive models with the airlines' passengers example.

A Classification Problem

The goal in classification problems is to assign previously unseen patterns to their respective classes, or categories, based on a set of representative pattern examples from each class. For example, Table 1.1 shows a set of points of the plane classified in two categories denoted by 0 and 1, respectively.

These two classes correspond to the half planes determined by the line shown in Figure 1.16.

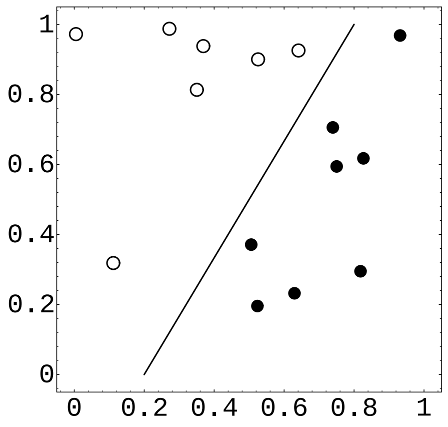


FIGURE 1.16. Linearly separable data sets.

We can train a perceptron with these points to automatically obtain the classification criterion. For example, if we consider a perceptron with two inputs, x_i and y_i, and one output c_i with linear activation function

$$c_i = w_1 x_i + w_2 y_i + q, \tag{1.19}$$

then we have a simple model containing three parameters w_1, w_2, and q. If we choose some initial values for the parameters and run the delta-rule algorithm with $\eta = 0.2$, then we obtain the RMS error curve shown in Figure 1.17. Figure 1.18 shows the evolution of the three parameters with the number of iterations. From the above figures, we can see how the optimization method reaches its optimum after 20 iterations.

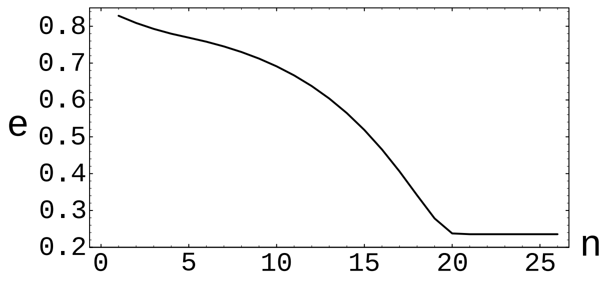


FIGURE 1.17. RMS error vs. the number of iterations n.

Finally, at convergence we get the model:

$$c_i = 1.28x_i - 0.815y_i + 0.384. \tag{1.20}$$

Figure 1.19 shows the values assigned by the function (1.20) to the points in $(0,1) \times (0,1)$. Values close to zero or one correspond to a high confidence

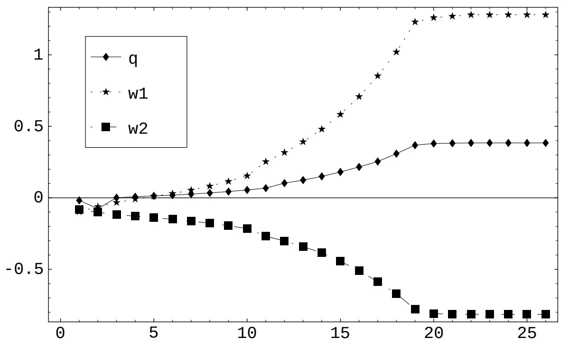


FIGURE 1.18. Evolution of the weight values w_1, w_2, and q as a function of the number of iterations n.

of the point being in the corresponding class, whereas intermediate values correspond to confuse situations.

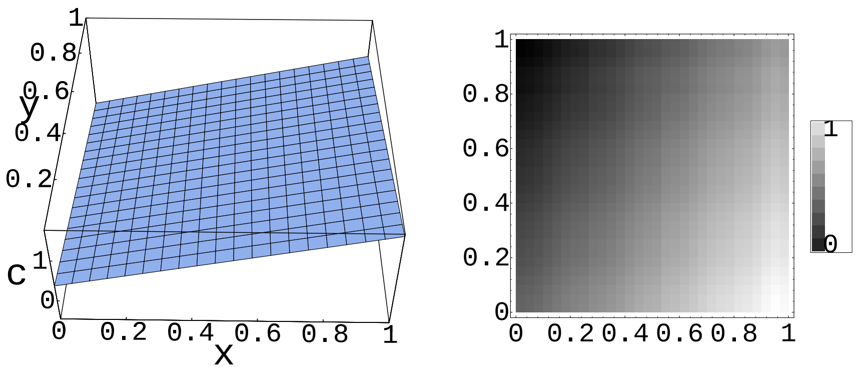


FIGURE 1.19. Linear classification surface (left) and density plot (right) for the points in $(0, 1) \times (0, 1)$ with values between 0 and 1.

If we consider a sigmoidal $f(x) = (1 + e^{-x})^-1$ or a step $\Theta(x)$ activation function in (1.19) and use the delta-rule algorithm to obtain a classification criterion, then we get the classification surfaces shown in Figures 1.20 and 1.21, respectively.

International Airlines' Passengers

A large number of prediction problems are concerned with estimating the next and future values of a time series $\ldots, x_{t-2}, x_{t-1}, x_t$. A standard technique to solve these problems consists of using autoregressive models where

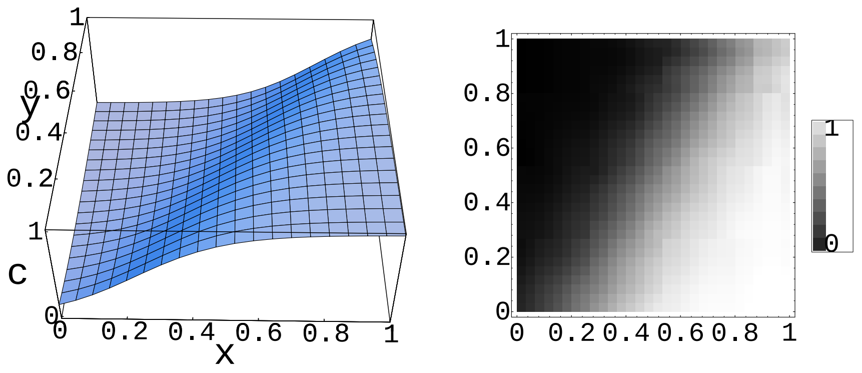


FIGURE 1.20. Sigmoidal classification surface (left) and density plot (right) for the points $(0, 1) \times (0, 1)$ with values between 0 and 1.

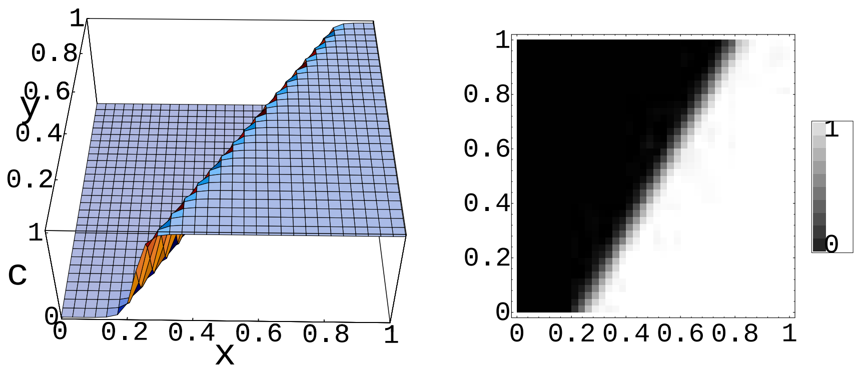


FIGURE 1.21. Step classification surface (left) and density plot (right) for the points $(0, 1) \times (0, 1)$ with values between 0 and 1.

the inputs are delayed values of the variable:

$$x_t = F(x_{t-1}, x_{t-2}, \ldots). \tag{1.21}$$

In this example we use the data given by Box and Jenkins (1976) consisting of a monthly time series of the number of passengers using international airlines from January, 1949 to December 1960. This time series presents some well-known properties of temporal series, such as non-linear trend and stationarity (see Figure 1.22). The aim of this example is to show that linear perceptrons are simple linear autoregressive models and, therefore, the power of these models can be incremented by considering nonlinear activation functions, leading to a special type of nonlinear autoregressive model. To create the training set we must first determine how many and which delayed outputs affect the next output. In this case, due to the nature of the time series we consider the lags, x_{n-1}, x_{n-12} and x_{n-24}, as inputs. After running 1000 iterations of the delta-rule algorithm we get the following model:

$$x_n = 0.025 + 0.12x_{n-1} + 0.71x_{n-12} + 0.25x_{n-24}.$$

The RMSE and maximum errors obtained in this case are 0.023 and 0.076, respectively, indicating a good quality of the resulting model. Figure 1.23 shows both the actual and predicted values of the time series (upper graph) and the errors obtained (lower graph).

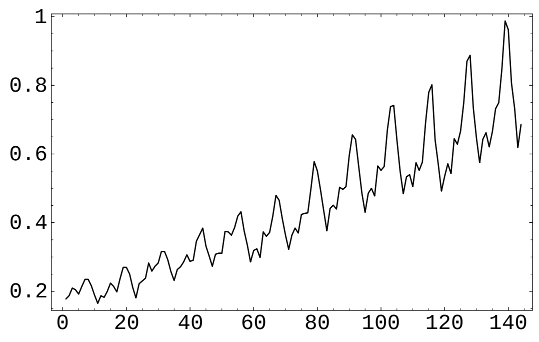


FIGURE 1.22. Airlines' passengers time series.

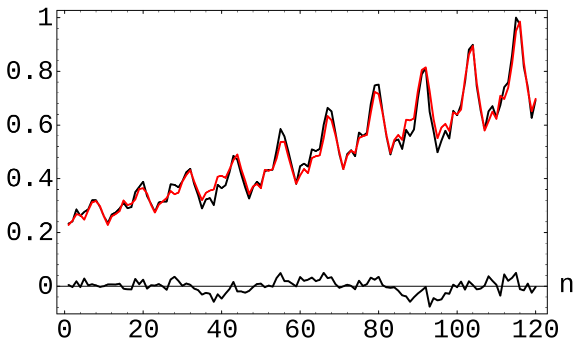


FIGURE 1.23. Results for the forecasted time series and error from a linear perceptron with three inputs and one output.

1.8 Multi-layer Perceptrons

In addition to an input and an output layer, a multi-layer perceptron has internal hidden layers. The nodes in the input layer feed the network distributing forward the input signals. Each node in the hidden and output layers receive an input from the nodes in the previous layer and computes an output value for the next layer. Figure 1.24 shows a two-layers perceptron. The addition of hidden layers give this type of architecture enough flexibility to solve many problems which the simple perceptrons can not.

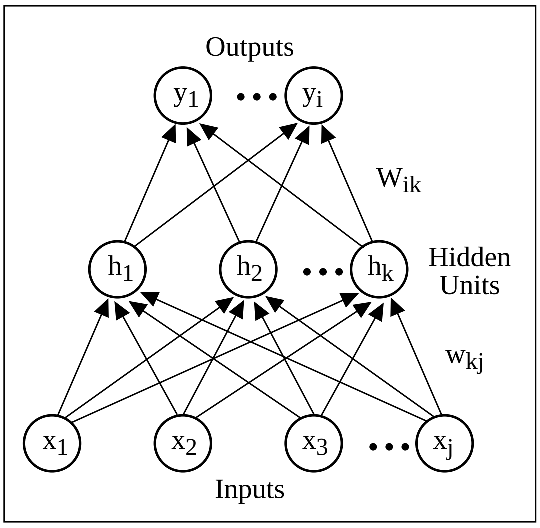


FIGURE 1.24. Two-layers $j : k : i$ perceptron.

For each of the outputs, y_i, a multi-layer perceptron computes a function $y_i = F_i(x_1, \ldots, x_n)$ on the inputs. For example, the neural network shown in Figure 1.24 defines the function

$$y_i = \sum_k f(W_{ik}\, f(\sum_j w_{kj} x_j - \theta_k) - \theta_i). \tag{1.22}$$

It has been shown that any set of functions $F_i(x_1, \ldots, x_n)$ can be approximated to a given accuracy by (1.22) when considering, at most, two hidden layers. When these functions are continuous then a single hidden layer is enough (see Cybenko (1989)). This solves partially one of the main shortcomings of multi-layer perceptrons: the design of an appropriate network structure for a given problem. Now, the problem reduces to choosing an appropriate number of hidden units to fit the model, but avoiding the overfitting problem. The solution to this problem is a trial and error procedure in most of the cases.

The most popular learning method for multi-layer perceptrons is known as *backpropagation* and is based on minimizing the sum of squared errors function by using a gradient descend method.

1.8.1 The Backpropagation Learning Algorithm

Suppose we have a set of inputs $\{a_{p1}, \ldots, a_{pn}\}$ and their corresponding outputs $\{b_{p1}, \ldots, b_{pm}\}$, $p = 1, \ldots, r$ as the training patterns. As in the case of the simple perceptron, we consider the sum of square errors function:

$$\begin{aligned} E(w) &= \frac{1}{2}\sum_{p,i}(b_{pi} - \hat{b}_{pi})^2 & (1.23)\\ &= \frac{1}{2}\sum_{p,i}(b_{pi} - f(\hat{B}_{pi}))^2 & (1.24)\\ &= \frac{1}{2}\sum_{p,i}(b_{pi} - f(\sum_k W_{ik}\hat{h}_{pk}))^2 & (1.25)\\ &= \frac{1}{2}\sum_{p,i}(b_{pi} - f(\sum_k W_{ik} f(\hat{H}_{pk})))^2 & (1.26)\\ &= \frac{1}{2}\sum_{p,i}(b_{pi} - f(\sum_k W_{ik} f(\sum_j w_{kj}a_{pj})))^2 & (1.27) \end{aligned}$$

The backpropagation algorithm is based on the same idea of gradient descending used in the delta-rule method. Then, the weights are iteratively moved down the slope of the error function (in the space of weights):

$$\Delta W_{ik} = -\eta\frac{\partial E}{\partial W_{ik}};\ \Delta w_{kj} = -\eta\frac{\partial E}{\partial w_{kj}}, \tag{1.28}$$

where the term η is the *learning parameter*, related to the rate of weight changing.

Since (1.23)-(1.27) involve the weights of both the hidden and output neurons, then the problem of iteratively updating the weights is not as simple as in the case of one-layer perceptrons. A solution to this problem was given with the two-steps backpropagation algorithm (see, for example, Rumelhart et al. (1986)). First, the input of a pattern a_p is propagated forward obtaining the value of the hidden, $\hat{h}_p$, and output, $\hat{b}_p$, units and, therefore, the associated error. The obtained values are used to update the weights W_{ik} of the output layer using (1.25). Afterwards, the obtained weights are used to update the weights of the hidden layer, w_{kj}, using (1.27) propagating the error backwards.

Let us first consider the case of the output layer. We use the chain rule as follows:

$$\Delta W_{ik} = -\eta\frac{\partial E}{\partial W_{ik}} = -\eta\frac{\partial E}{\partial \hat{b}_{pi}}\frac{\partial \hat{b}_{pi}}{\partial \hat{B}_{pi}}\frac{\partial \hat{B}_{pi}}{\partial W_{ik}}, \tag{1.29}$$

where $\hat{b}_{pi} = f(\hat{B}_{pi})$ is the network i-th output obtained by forward propagating the input $(a_{p1}, \ldots, a_{pn})$. Then, we have

$$\begin{aligned} \frac{\partial E}{\partial \hat{b}_{pi}} &= -(b_{pi} - \hat{b}_{pi}), \\ \frac{\partial \hat{b}_{pi}}{\partial \hat{B}_{pi}} &= f'(\hat{B}_{pi}), \\ \frac{\partial \hat{B}_{pi}}{\partial W_{ik}} &= \hat{h}_{pk}. \end{aligned}$$

Then, we obtain the following updating rule:

$$\Delta W_{ik} = \eta\, \hat{h}_{pk}\, \delta_{pi}, \text{ where } \delta_{pi} = (b_{pi} - \hat{b}_{pi}) f'(\hat{B}_{pi}). \tag{1.30}$$

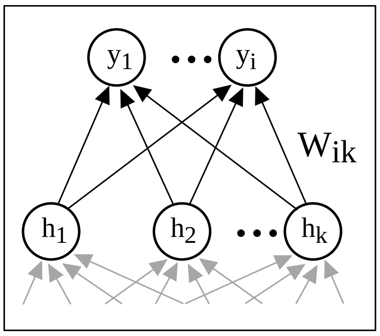


FIGURE 1.25. Obtaining the weights of the output unit using the values $\hat{b}_{pi}$, $\hat{h}_{pk}$ obtained from the network.

Once the output weights have been updated, the resulting value together with the input, hidden and output values are used to modify the weights in the hidden layer:

$$\Delta w_{kj} = -\eta \frac{\partial E}{\partial w_{kj}} = -\eta \sum_i \frac{\partial E}{\partial \hat{b}_{pi}} \frac{\partial \hat{b}_{pi}}{\partial \hat{B}_{pi}} \frac{\partial \hat{B}_{pi}}{\partial \hat{h}_{pk}} \frac{\partial \hat{h}_{pk}}{\partial \hat{H}_{pk}} \frac{\partial \hat{H}_{pk}}{\partial w_{kj}}. \tag{1.31}$$

The first two terms were obtained before. For the rest we have:

$$\begin{aligned} \frac{\partial \hat{B}_{pi}}{\partial \hat{h}_{pk}} &= W_{ik}, \\ \frac{\partial \hat{h}_{pk}}{\partial \hat{H}_k} &= f'(\hat{H}_k), \\ \frac{\partial \hat{h}_{pk}}{\partial w_{kj}} &= a_{pj}, \end{aligned}$$

$$\Delta w_{kj} = \eta\, a_{pj}\, \psi_{pk} \qquad \text{where } \psi_{pk} = \sum_i \delta_{pi}\, W_{ki}\, f'(H_{pk}). \tag{1.32}$$

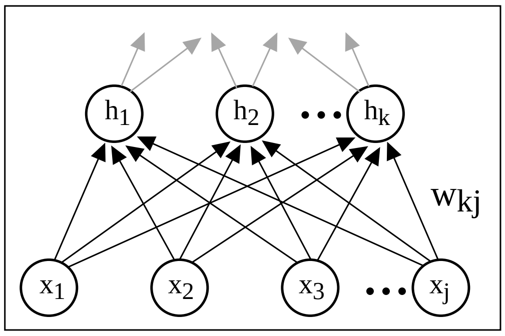


FIGURE 1.26. Obtaining the weights of the output unit using the values $\hat{b}_{pi}$, $\hat{h}_{pk}$, obtained from the network, and W_{ik}.

1.8.2 Backpropagation Algorithm

In the following algorithm, and with the aim of avoiding formal derivatives, we shall consider sigmoidal activation functions:

1. Initialize the weights to random values.
2. Choose a training pattern and propagate it forward obtaining the values $\hat{h}_{pk}$ and $\hat{b}_{pi}$ for the neurons in the hidden and output layers.
3. Calculate the error associated with the output units:
$$\delta_{pi} = (b_{pi} - \hat{b}_{pi}) f'(\hat{B}_{pi}) = (b_{pi} - \hat{b}_{pi})\hat{b}_{pi}(1 - \hat{b}_{pi}).$$
4. Calculate the error associated with hidden units:
$$\psi_{pk} = \sum_i \delta_{pi}\, W_{ki}\, f'(\hat{H}_{pk}) = \sum_i \delta_{pi}\, W_{ki}\, \hat{h}_{pk}(1 - \hat{h}_{pk}).$$
5. Calculate:
$$\Delta W_{ik} = \eta\ \delta_{pi}\, \hat{h}_{pk}\,,$$
and
$$\Delta w_{kj} = \eta\, \psi_{pk}\, a_{pj}\ ,$$
and update the weights according to the obtained values.
6. Repeat the above steps for each of the training patterns.

1.9 Feed Forward Neural Network Examples

In this section we analyze several illustrative examples using multi-layer perceptrons. We shall use several programs[7], which are intended to be used as a tool for beginners and provide an alternative to using more expensive and hard to use packages.

1.9.1 A Nonlinear Classification Problem

In Example 1.7.3 we used a single-layer perceptron to solve a linear separable problem (two sets of points separated by a line in the plane). It has been shown that this is the most general type of problems that simple perceptrons can deal with (see Minsky and Papert (1969)). For example, suppose we have a set of 100 random points of the interval $(-1, 1)$ classified in two categories: those inside the circle shown in Figure 1.27 (filled dots), and those outside (empty dots).

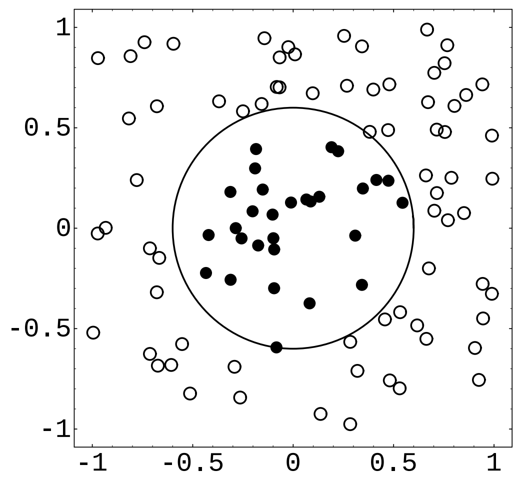


FIGURE 1.27. Set of data points separated in categories by the circle.

Since these two categories are not linearly separable, then we need a multi-layer network with, at least, one hidden layer to classify both groups. In this example, if we consider a single hidden layer, then we need five or more hidden neurons to have a stable model to fit the data. If we only

[7]There are several available shareware programs which implement multi-layer feed forward neural networks and incorporate user friendly interfaces. For example the program *NNModel*, from the web site *http://www.neuralfusion.com/*, and *Neural Network Development Tool*, from *http://www.abo.fi/˜bjsaxen/nndt.html.*

consider 5 hidden neurons then the convergence will depend on the initial weight configuration (multiple local minima). Figure 1.28 illustrates this fact by considering two different initial weight configurations. In one of the cases, the backpropagation algorithm gets stuck on a local minima; however, in the other case it converges to the optimal minimum. Figure 1.29 shows the classification surface obtained from the neural network in this case[8].

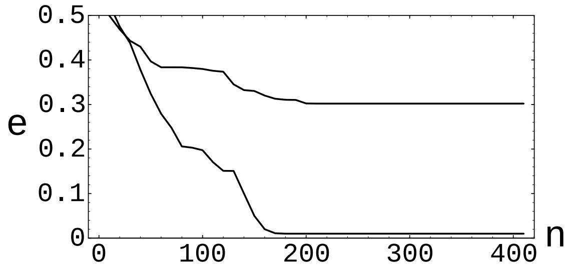


FIGURE 1.28. Convergence of the RMS error starting from two different weight configurations.

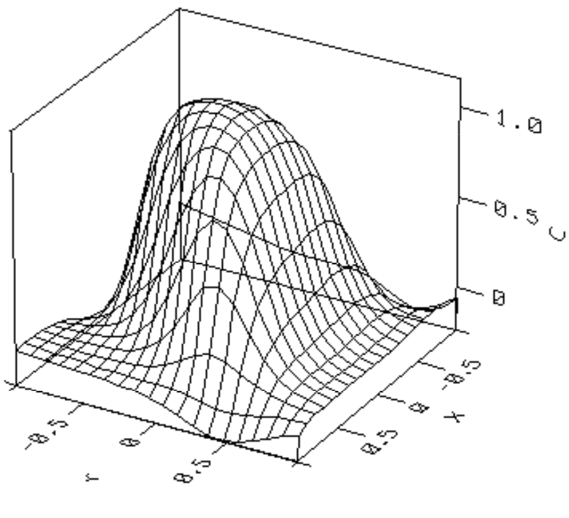


FIGURE 1.29. Classification surface obtained from a 2 : 5 : 1 neural network.

[8]All the calculations in this example have been performed with the program *NNModel* available at the web site *http://www.neuralfusion.com/*. For instance, Figure 1.29 is a snapshot of a program's window.

x	y	x	y	x	y	x	y
-3.002	-0.032	-2.952	0.005	-2.892	0.077	-2.785	0.134
-2.600	0.078	-2.557	0.100	-2.476	0.028	-2.301	0.220
-2.207	0.2396	-1.922	0.500	-1.792	0.649	-1.781	0.671
-1.709	0.748	-1.607	0.887	-1.469	1.042	-1.088	1.170
-0.908	0.993	-0.716	0.954	-0.635	0.811	-0.122	0.868
-0.1041	0.891	0.060	1.059	0.221	1.140	0.2984	1.122
0.368	1.121	0.749	0.490	0.889	-0.027	1.166	-0.542
1.265	-0.798	1.326	-0.832	1.400	-0.748	1.446	-0.859
1.787	-0.586	1.848	-0.528	1.892	-0.484	2.066	-0.323
2.090	-0.309	2.315	-0.224	2.421	-0.109	3.005	0.014

TABLE 1.2. Set of training points (x, y) sampled from the function $g(x)$ with added Gaussian noise.

1.9.2 Estimating a Function From Data

An interesting problem with many applications consists of estimating a function from a sample set of input-output pairs and with no knowledge of the form of the function. This problem is generally known as nonparametric learning and it consists of approximating the function using a functional form which allows a wide class of functions to be represented (polynomial functions, Fourier series, etc.). Neural networks can be viewed as a special type of nonparametric models which implement sigmoidal functions. As we have seen before, a feed forward neural network with at least one hidden layer can approximate any nonlinear function to a given degree of accuracy. It is important to remark here that the resulting weights have no particular meaning in relation to the problem to which they are applied (they are only the coefficients of the arbitrary family of sigmoidal functions implemented by the neural network).

For example consider the set of 40 training input-output points sampled from the function $g(x) = (1 + x - x^2)e^{-x^2}$ with added Gaussian noise with standard deviation $\sigma = 0.05$. The data are shown in Table 1.2 and in Figure 1.30.

We have used a two-layers perceptron with different number of hidden units to approximate the noisy data. Table 1.3 shows the RMS and maximum errors obtained when using the nonlinear function $g(x)$ in three neural networks with 5, 10, and 15 hidden units, respectively, to fit the noisy data. From this table it can be seen that the error decreases when increasing the number of hidden neurons. Moreover, the largest neural model (15 hidden units) fits the noisy data better than the original model, meaning that it is fitting both the model and the noise (overfitting). Figures 1.31 and 1.32 shows the approximated functions associated with the small and large neural models.

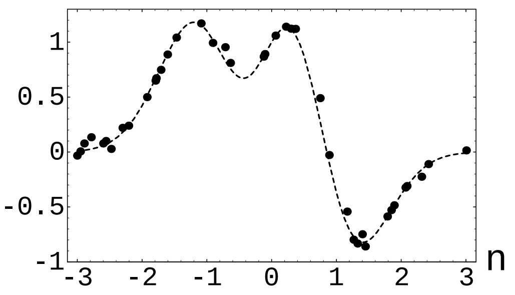


FIGURE 1.30. The function $g(x) = (1 + x - x^2)e^{-x^2}$ and a set of sampled points with added Gaussian noise.

	$g(x)$	1:5:1 NN	1:10:1 NN	1:15:1 NN
RMSE	0.056	0.085	0.062	0.044
Max	0.212	0.246	0.193	0.139

TABLE 1.3. RMS and maximum errors obtained when using the nonlinear function $g(x)$, a three different neural networks to fit the noisy data.

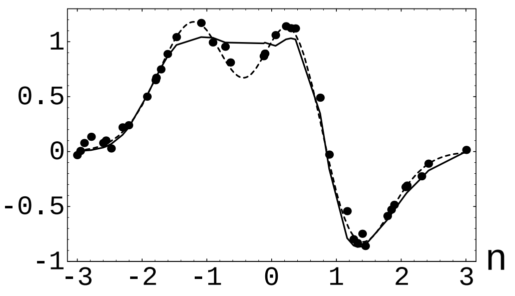


FIGURE 1.31. Original nonlinear function (dotted line) and approximated function obtained from a neural network with five hidden units (filled line).

1.9.3 Electric Power Consumption

Consumption of electric power is one of the main indicators of economic activity. In this example we use the data given by Pandratz (**?**) consisting of a 180-points time series to predict the electric power consumption as a function of three other variables: *time*, *cool*, and *heat*. The variable *time* indicates the month of the year. The variables *cool* and *heat* measure the energetic efforts to compensate the increases and decreases of the temperature. The variable *heat* is defined as 65 Farenheit degrees minus the temperature averaged on the month, in the case this temperature is below 65 degrees (heating need), and zero, otherwise (see Figure 1.33(b)). On the

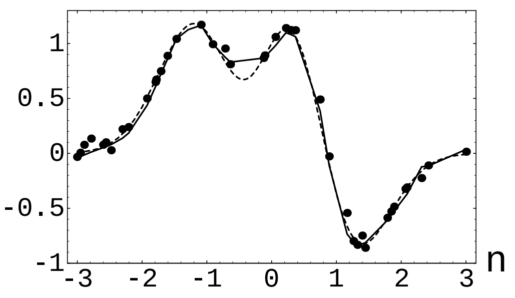


FIGURE 1.32. Original nonlinear function (dotted line) and approximated function obtained from a neural network with fifteen hidden units (filled line).

other hand, the variable *cool* is defined as 65 Farenheit degrees plus the temperature averaged on the month, in the case this temperature is above 65 degrees (cooling need), and zero, otherwise (see Figure 1.33(c)). The variable *consumption* indicates the consumption of electric power (Kw/h) from January 1970 to December 1984 (represented by a month number in the variable *time*) (see Figure 1.33(a)). It is obvious that the fluctuations in this variable can be associated with changes in the temperature. Note that there is a sudden change in the electric power consumption series corresponding to the end of 1973, coinciding with the start of the petroleum crisis (October-November 1973). This corresponds to the last 10 points in the time series.

With the aim of investigating the performance of different neural networks on this set of data, we have trained a two-layer perceptron with 3 inputs, one output and different numbers of hidden neurons. A good approximation of the time series can be obtained by considering only three hidden neurons. The results of a $10,000$ iterations training are shown in Figure 1.34. In this case the average squared error obtain for 10 different initial weight configurations was 0.163.

1.9.4 Chaotic Time Series: The Hénon Map

The Hénon map is one of the most illustrative examples of a simple nonlinear system exhibiting the complex dynamics known as deterministic chaos, and can be defined in the following way:

$$x_n = 1 - ax_{n-1}^2 + 0.3x_{n-2}. \tag{1.33}$$

It can be shown that, depending on the values of the parameter a, the system exhibits a very rich behavior, including the presence of deterministic chaos. In particular we shall consider the chaotic dynamical system defined by $a = 1.4$. A time series plot of the Hénon map for the initial conditions $x_0 = 0.5, x_1 = 0.5$ is given in Figure 1.35. This figure shows the seemingly

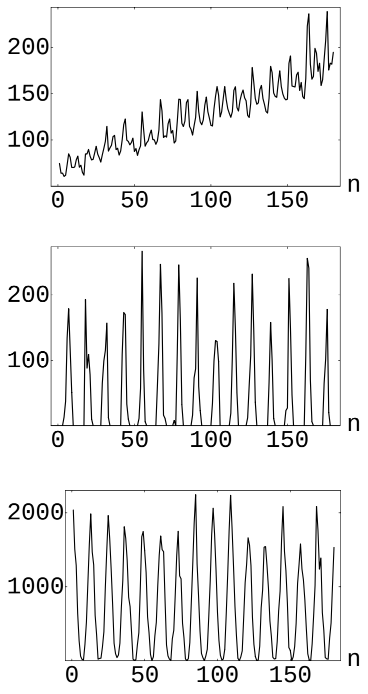


FIGURE 1.33. Time series for the consumption, heat and cool variables.

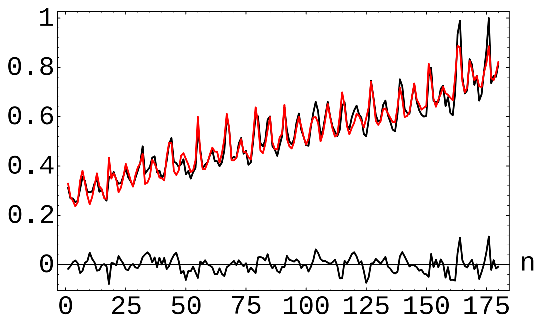


FIGURE 1.34. Approximation and errors obtained with a 3:3:1 neural network.

stochastic dynamics of the system that is due to the sensitivity on the initial conditions characteristic of chaotic maps.

However, the delay space (x_{n-2}, x_{n-1}, x_n) of the time series shows the deterministic structure of the underlying process (Fig. 1.36).

Suppose that we have the time series consisting of 80 points shown in Figure 1.35 and that we want to obtain a representative model of the under-

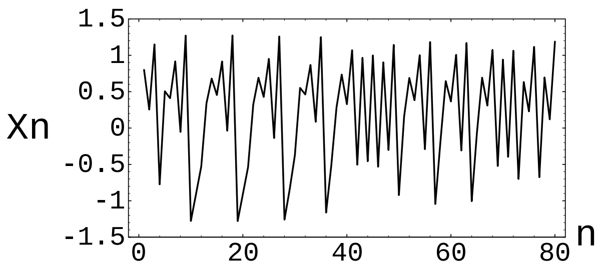


FIGURE 1.35. Time series for the Hénon map.

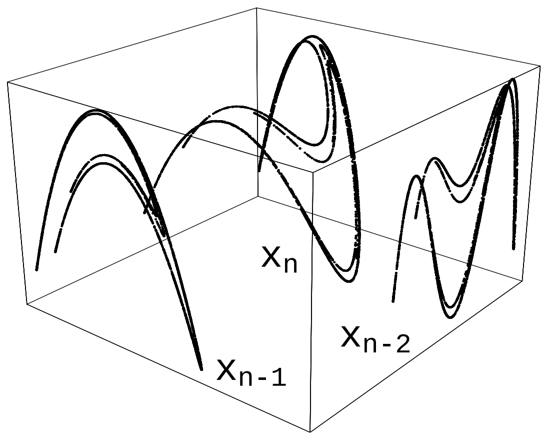


FIGURE 1.36. Phase space of the Hénon map.

lying dynamics. Then, we can use as output the values of the time series x_n and as inputs the two delayed coordinates x_{n-1} and x_{n-2} in (1.33). We can use a 2:5:1 two-layer perceptron (21 parameters) trained with the back-propagation algorithm using the same Hénon time series data and performing $10,000$ iterations. The RSME error obtained was 0.009 and the maximum error 0.03. Moreover, when increasing the number of hidden neurons to 10 (41 parameters) the RMSE and maximum error only decrease to 0.006 and 0.02, respectively. On the other hand, if we consider a smaller number of hidden neurons, then the model will fit the data poorly. This can be easily shown by using one of the programs mentioned above. For example, Figure 1.37 shows the main window of the NNdt program displaying a 2:3:1 neural network. With the aim of validating the model, a test data consisting of the next 100 points of the orbit was considered. Figure 1.38 shows the convergence of both the training and test errors as a function of the number of iterations, and Figure 1.39 compares the original and approximated models. The above analysis indicates that the network with 5 hidden units is a good approximate model of the Hénon time series.

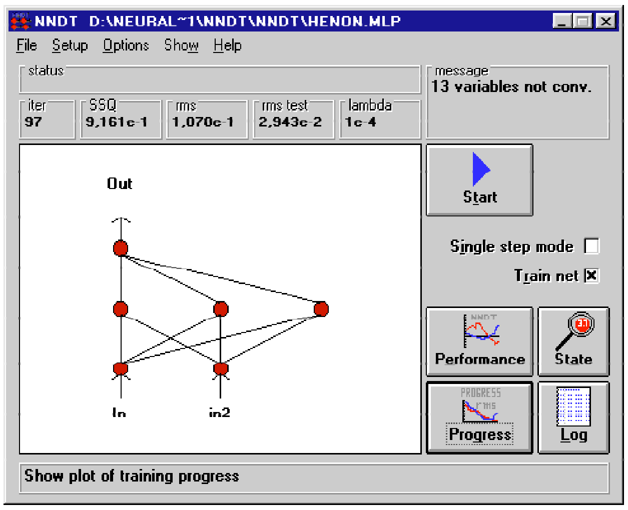


FIGURE 1.37. Main window of the NNdt program showing a 2:3:1 perceptron.

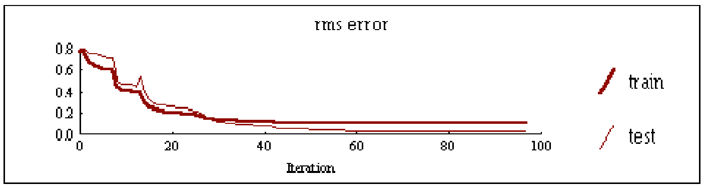


FIGURE 1.38. RMS training and test error as a function of the number of iterations.

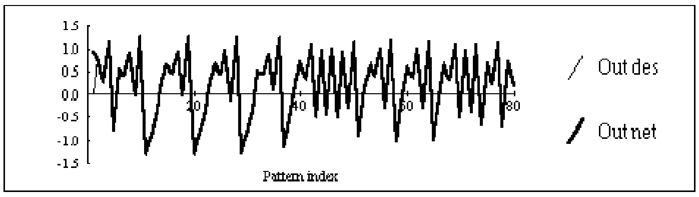


FIGURE 1.39. The Hénon time series ("out des" in the figure) and the corresponding network output (out net) are overlapped indicating a good approximation.

In some cases the optimization process may converge to a local minimum and not to the global one. This is due to the large number of parameters involved in the neural network and to the learning algorithm characteristics.

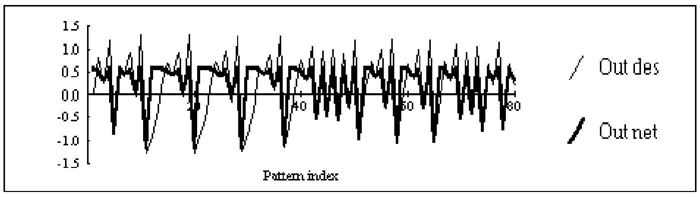


FIGURE 1.40. The network output settles on a stable non-optimal configuration corresponding to a local minimum.

1.10 Competitive Neural Networks

Competitive networks are a popular type of unsupervised network architectures which are widely used to automatically detect clusters, or categories, within the available data. A simple competitive neural network is formed by an input and an output layer, connected by feed forward connections (see Figure 1.41). Each input pattern represents a point in the configuration space (the space of inputs) where we want to obtain classes. Then, the output layer contains as many neurons as classes, or categories, we want to obtain. The competitive dynamics among the classes is created by including lateral connections among the neurons in the output layer. Each output neuron is connected to itself by an *excitatory* positive weight and to all others by *inhibitory* negative connections.

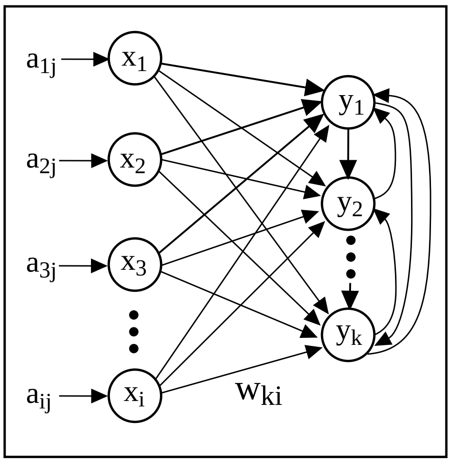


FIGURE 1.41. A simple competitive neural network.

This type of architecture is usually trained with a *winner takes all* algorithm, so that only the weights associated with the output neuron with largest value (the winner) are updated. Consider the training data consisting of a set of input patterns $(a_{1j}, \ldots, a_{nj}), j = 1, \ldots, m$. As in the previous network architectures, we start with small random values for the weights. Then, the input patterns are applied in turn to the network, say we apply pattern $(a_{1j}, \ldots, a_{nj})$, and the winner output unit is selected, say y_k. To make this neuron more likely to win on the given input pattern, the weights associated with this neuron are modified according to

$$\Delta w_{ki} = \eta(a_{ij} - w_{ki}). \tag{1.34}$$

The effect of this change is moving the weight vector $(w_{k1}, \ldots, w_{kn})$ directly towards $(a_{1j}, \ldots, a_{nj})$.

Once the network has been trained, the winner output associated with a given pattern indicates the associated cluster.

Note that in the case of supervised classification using multi-layer perceptrons, the user provides examples of the different categories. However, this new architecture provides a classification method which automatically detects groups of similar patterns in the configuration space.

More general competitive architectures has been proposed to avoid the problem of working with a fixed number of output units (classes). For instance, the Adaptive Resonance Theory (ART) introduced by Grossberg uses the network's sensitivity to detail within the training data to determine the optimal number of outputs needed to classify the data (see Grossberg (1976) for a complete description of this model).

Example 1.1 (Competitive networks) Consider the set of points generated from three different independent bivariate Gaussian distributions with variance $\sigma^2 = 0.08$ shown in Figure 1.42. The shaded areas in the figure indicate two standard deviations of the corresponding Gaussian distributions.

In this example we shall illustrate the use of competitive neural networks to cluster the data set in a given number of categories. In this example we have two inputs, the coordinates x and y, and we shall consider two different networks with two and three output units, respectively (see Figure 1.43).

The weight vectors resulting from the networks when trained with the above data are shown in Figures 1.44. Figure 1.44(a) shows the vectors $(w_{1i}, w_{2i}), i = 1, 2$ corresponding to the two output units. Similarly, Figure 1.44(b) shows the three associated vector weights. Note that, in the case of two output units, the points corresponding to the Gaussian distributions shown in the upper part of Figure 1.42 are assigned the same class. However, when increasing the number of categories, i.e., the number of output units, each of the three different distributions is assigned to a different cluster.

Exercises

1.1 Determine the number of parameters of a multi-layer perceptron with k hidden units, j input units and i output units.

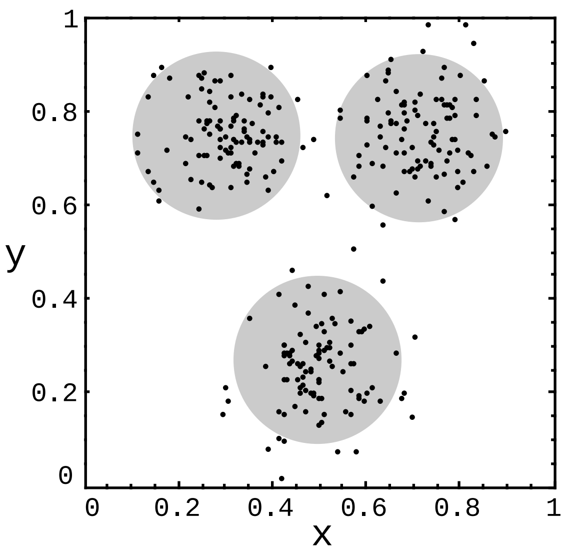


FIGURE 1.42. Random data consisting of 300 points generated from three different independent bivariate Gaussian distributions with variance $\sigma^2 = 0.08$ and means $(0.3, 0.75)$, $(0.5, 0.25)$, and $(0.75, 0.75)$, respectively.

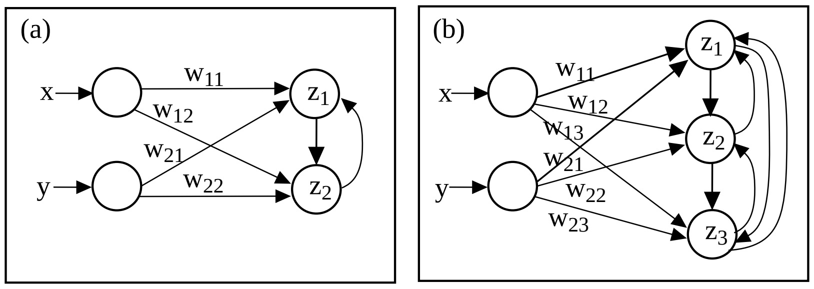


FIGURE 1.43. Two competitive neural network with (a) two and (b) three output units.

1.2 Show that for networks with *tanh* or *logistic* sigmoidal hidden unit activation function, the network mapping is invariant if all of the weights and the bias feeding into and out of a unit have their signs changed.

1.3 A criterion for deciding if two sets of points are linearly separable is as follows: define the convex set generated by a set of points $X = \{\mathbf{x}_i\}$ as

$$\{\sum_i \alpha_i \mathbf{x}_i \mid \sum_i \alpha_i = 1 \text{ and } \alpha_i \geq 0 \text{ , } \forall\, i\}.$$

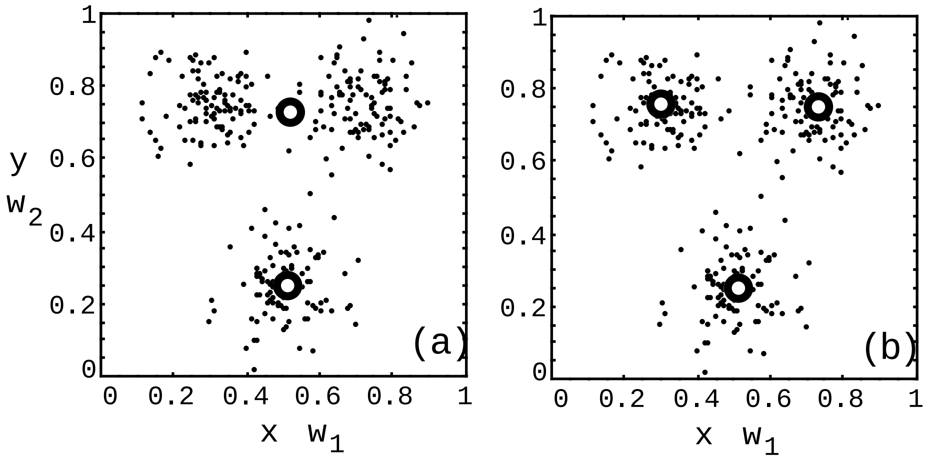


FIGURE 1.44. Examples of competitive learning with (a) two and (b) three output units. The circles show the weight vector of each unit after training on these data.

Let $Y = \{\mathbf{y}_i\}$ be a second set of points and its corresponding convex set. The sets X and Y are linearly separable if there exists a vector $\mathbf{v}$ and a scalar v_0 such that $\mathbf{v}^T\mathbf{x}_i + v_0 > 0$ for all $\mathbf{x}_i \in X$, and $\mathbf{v}^T\mathbf{y}_i + v_0 < 0$ for all $\mathbf{y}_i \in Y$. Prove that X and Y are linearly separable iff their convex sets do not intersect.

1.4 Write a software implementation of the backpropagation algorithm.

1.5 Add a Gaussian noise with standard deviation 0.05 to the function $y = 0.5 + 0.3\cos(2\pi x)$. Generate a set of 50 points (x, y), approximate the data with neuronal networks and plot the exact and the approximate solution.

1.6 Table 1.4 shows the coordinates of 15 points (x, y) on the plane, classified in two different categories, $c = 0$ or 1.

(a) Plot the points on the plane and separate the two categories using a straight line.

(b) Look for a classification criterion.

(c) Design a neural network to classify points in the two categories, and obtain the RMSE.

1.7 Approximate the data points in Table 1.5 by a function using neuronal networks:

x	y	c	x	y	c	x	y	c
0.432	0.321	1	0.332	0.021	0	0.325	0.543	1
0.512	0.225	1	0.087	0.211	0	0.125	0.443	0
0.212	0.121	0	0.234	0.412	1	0.045	0.521	0
0.887	0.211	1	0.150	0.365	0	0.732	0.352	1
0.234	0.112	0	0.245	0.521	1	0.082	0.352	0

TABLE 1.4. Data points (x, y)

x	y	x	y
0	0.009	0.50	0.4128
0.05	0.1187	0.55	0.4389
0.10	0.2121	0.60	0.4983
0.15	0.4182	0.65	0.3486
0.20	0.4671	0.70	0.4423
0.25	0.4708	0.75	0.4596
0.30	0.6383	0.80	0.4591
0.35	0.5660	0.85	0.5579
0.40	0.5251	0.90	0.7929
0.45	0.4489	0.95	0.9137

TABLE 1.5. Data points (x, y)

x	y	c	x	y	c	x	y	c
1	0.600	0	0.641	0.231	1	0.121	0.098	1
0.900	0.232	1	0.436	0.123	1	0.436	0.723	1
0.632	0.531	0	0.111	0.198	0	0.723	0.691	0
0.456	0.323	0	0.200	0.100	1	0.220	0.530	1
0.200	0.200	0	0.621	0.832	1	0.121	0.598	1
0.723	0.291	1	0.723	0.991	1	0.883	1	1

TABLE 1.6. Data points (x, y), and category $(c = 0, 1)$.

(a) With 2 hidden units.

(b) With 3 hidden units.

(c) With 4 hidden units.

(d) Calculate the RMSE in all cases and compare them.

1.8 Use a multilayer network for obtaining a classification criterion for the points in Table 1.6

1.9 Use a Hopfield network to store and recover the images in Figure 1.45.

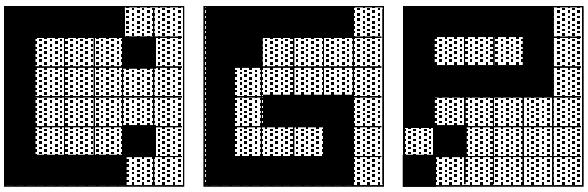


FIGURE 1.45. Pattern to be stored.

1.10 Consider the Hénon map $x_n = 1 - 1.5x_{n-1}^2 + 0.3x_{n-2}$, then

(a) Plot the time series with the initial conditions $x_0 = 0.2$ and $x_1 = 0.2$.

(b) Generate a noisy time series by adding a random white noise to the obtained Hénon series.

(c) Approximate both sets of data using a feed forward network.

(d) Calculate the error in both cases.

Part II

FUNCTIONAL NETWORKS

This part is devoted to functional networks. In it, all the elements required for working with this type of networks are given. In Chapter 2 we first give a simple introduction to functional networks, by means of two illustrative examples: the medical diagnosis and the Bayesian conjugate distributions problems. With these examples, we aim to:

1. Show how the physical or engineering properties of the problem being considered allow obtaining the initial topology (structure) of the functional network. In other words, how the initial topology of the network is related to the problem itself.

2. How functional equations allow simplifying not only the initial topology but the neuron functions, which can be expressed in terms of functions of less arguments.

3. Illustrate the uniqueness problem, its relation to the learning problem, and show how it can also be solved with the help of functional equations.

4. Introducing the learning methods based on minimizing an error function which measures the differences between the desired and the actual outputs.

Next, we describe the different components of a functional network and compare them with neural networks, discussing their main analogies and differences.

Once the need of functional equations has been made clear and the reader is motivated for a deep analysis of them, they are studied in Chapter 3. We start with a definition of functional equations and some simple examples from geometry, economics, civil engineering, statistics, etc. Next, a list of practical methods for solving functional equations is given, each with an illustrative example.

In the last part of the chapter, some functional equations in functions of a single and several variables, which have applications to functional networks, are analyzed and their solutions are given.

In Chapter 4 some important functional networks models are presented. For each model, the physical meaning of its initial topology is given; thus, suggesting the types of problems it is adequate to. Then, they are simplified, their uniqueness problem solved and an estimation method given. This allows the user to have a starting list of models to work with, either using them because of their physical meaning or as black boxes. Some simple examples illustrate the use of these models.

Finally, in Chapter 5 we deal with the problem of model selection. Since many different network topologies and approximations of the neuron functions are possible, one method for selecting the optimum model is required. Though other alternative methods are mentioned, the minimum description length principle is used to propose a quality measure to be used in the selection procedure.

Since in many practical cases, exhaustive searching for the optimum model is infeasible (because of the complexity of the problem), four alternative methods are proposed: the backward, forward, backward-forward and forward-backward methods. Some examples of applications are also presented.

Chapter 2
Introduction to Functional Networks

2.1 Introduction

In Chapter 1 we have shown that neural networks are powerful tools to solve a wide variety of practical problems. Neural networks consist of one or several layers of neurons connected by links. Each neuron computes a scalar output from a weighted combination of inputs, coming from the previous layer, using a given scalar activation function (usually step or sigmoidal functions). Since the neural functions are given, the parameters of the neural network are the weights of the connections. Then, learning consists of obtaining the optimal weights to reproduce a given set of data.

Despite of its wide diffusion and extensive application in several domains, the standard neural networks paradigm has some limitations and has been extended in several directions. Examples of such extensions are the higher order networks (Lee et al. (1986)), the probabilistic neural networks (see Specht (1990)), the fuzzy neural networks (see Gupta and Rao (1994)) and the wavelets neural networks (see Zhang (1992)). However, all them treat neural networks as black boxes and do not take into account the functional structure and properties of the function being modeled (domain knowledge).

In this chapter we introduce *functional networks*, a novel generalization of neural networks that brings together domain knowledge, to determine the structure of the network, and data, to estimate the unknown neuron functions. In functional networks not only arbitrary neural functions are allowed, but they are initially assumed to be multiargument and vectorvalued functions, i.e., to depend on several arguments and to be multivariate. An important characteristic of functional networks is the possibility of dealing with functional constraints determined by functional properties we may know about the model (e.g., associativity, distributivity, etc.). This can be done by considering coincident outputs (convergent links) of some selected neurons. This allows writing the value of these output units in several different forms (one per different link) and leads to a system of functional equations, which can be directly written from the topology of the neural network. Solving this system of functional equations leads to a great simplification of the initial network topology and neuron functions, in the sense that some of the neurons can be removed or simplified (their associated number of arguments is reduced). In fact, the initial multidimensional functions can be written in terms of functions of fewer arguments, which in many cases become single argument functions.

In functional networks there are two types of learning to deal with domain and data knowledge:

1. *Structural learning*, which includes:

   (a) the initial topology of the network, based on some properties which are available to the designer, and

   (b) the posterior simplification using functional equations, leading to a simpler equivalent architecture.

2. *Parametric learning*, concerned with the estimation of the neuron functions. This can be done by considering linear combinations of given functional families and estimating the associated parameters from the available data. Note that this type of learning generalizes the idea of estimating the weights of the connections in a neural network.

This chapter gives an overview of functional networks. In Section 2.2 functional networks are motivated by means of two simple examples. In Section 2.3 the structure of functional networks is analyzed, giving a detailed description of its main elements. In Section 2.4 we show the main differences between functional and neural networks. In Section 2.5 we motivate and describe the main steps of the process of working with functional networks, using an illustrative simple example.

2.2 Motivating Functional Networks

To motivate functional networks we use two simple examples associated with medical diagnosis and Bayesian conjugate distributions problems, respectively.

Example 2.1 (Medical diagnosis) Suppose that we wish to know the probability of a given patient to have a disease based on three continuous symptoms X, Y and Z, which take numerical values x, y and z, respectively. Assume that such a probability is $Q(x, y, z)$, that is, a function of the numerical values of the measured symptoms.

In addition, we make the assumption that the information given by the values of a pair of symptoms from the triplet $\{X, Y, Z\}$ about $Q(x, y, z)$ can be collected in a single value which is a function of such a pair. Similarly, this new value can be combined with the value of the remaining symptom, by another function, to give the value of $Q(x, y, z)$. More precisely, we assume that functions F, G, H, K, L and M exist such that

$$Q(x, y, z) \equiv F[G(x, y), z] = K[x, N(y, z)] = L[y, M(x, z)]. \tag{2.1}$$

Thus, the probability $Q(x, y, z)$ of the patient having the disease can be calculated by means of a function F of $G(x, y)$ (a summary of the influences of symptoms X and Y) and the value z associated with symptom Z. In other words, if at a given time, we know the values associated with symptoms X and Y, we can calculate the influence of these two symptoms on Q by means of $G(x, y)$, and later incorporate the influence of symptom Z by means of function F. The same argument is assumed to be valid for any permutation of the X, Y and Z symptoms.

If the variables measure symptoms which are directly related to the disease, it is reasonable to assume that the functions F, K, L, G, N and M are invertible (strictly monotonic) with respect to both variables. This means that the higher (lower) level of one symptom the higher (lower) probability of the disease.

Equations (2.1) suggest the network in Figure 2.1(a), where I is used to refer to the identity function $I(x) = x$ and the three convergent arrows in the unit u are used to indicate coincident values, i.e., the values coming from each of the links must be the same. However, this network is not a neural network because:

1. The neuron functions are arbitrary.
2. Some neuron functions are multiargument (e.g., F, G, K, L, M and N).
3. The outputs of neurons F, L and K are coincident, i.e., leading to the same output u.

Thus, neural networks are clearly inappropriate to reproduce this model.

The coincident connections in unit u or, equivalently, Equations (2.1) put strong conditions on the neuron functions F, G, K, L, M and N. The methods from the field of functional equations allows us to work with these equations and obtain the corresponding functional conditions (for a general

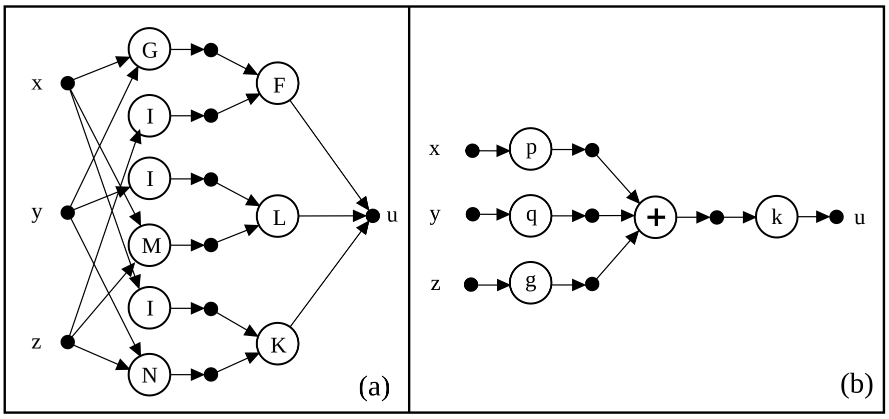


FIGURE 2.1. Equivalent functional networks resulting from a diagnosis problem.

introduction to functional equations see Aczél (1966) and Castillo and Ruiz-Cobo (1992)). For instance, as we shall see in the next chapter, the general continuous solution of the system of functional equations (2.1) is

$$\begin{array}{rcl} F(x,y) & = & k[f(x)+g(y)], \\ G(x,y) & = & f^{-1}[p(x)+q(y)], \\ K(x,y) & = & k[p(x)+n(y)], \\ N(x,y) & = & n^{-1}[q(x)+g(y)], \\ L(x,y) & = & k[q(x)+m(y)], \\ M(x,y) & = & m^{-1}[p(x)+g(y)], \end{array} \tag{2.2}$$

where $k.g, p$ and q are arbitrary functions and f, m and n are arbitrary invertible functions. The first four equations in (2.2) are the solutions of the first functional equation in (2.1), and the last four equations in (2.2) are the solutions of the last functional equation in (2.1).

Replacing this into (2.1) we get

$$Q(x,y,z) = k[p(x)+q(y)+g(z)]. \tag{2.3}$$

Equation (2.3) shows that the functional networks in Figures 2.1(a) and (b) are equivalent, in the sense of giving the same outputs for any given input. However, note that the network topology and the functional structure of the functional network shown in Figure 2.1(b) are much simpler than those associated with Figure 2.1(a).

Example 2.2 (Bayesian Conjugate Distributions) Suppose that the random variable X belongs to a uniparametric family of distributions with likelihood function $L(x,\theta)$, where $\theta \in \Theta$ is the parameter. Suppose also that, as Bayesians, θ has an "a priori" probability density function $F(\theta; y)$, i.e., it belongs to a uniparametric family with hyperparameter y. Then, Bayes' theorem guarantees that the "posterior" density is proportional to $L(x;\theta)F(\theta;y)$. If we force the "a posteriori" distribution to belong to the same family, we get the functional equation

$$F\left(\theta; G(x;y)\right) = L(x;\theta)F(\theta;y), \tag{2.4}$$

where G gives the value of the new parameter, as a function of the sample value x and the old hyperparameter value y.

Equation (2.4) suggests the network in Figure 2.2(a), that is not a neural network, because:

1. The neuron functions are different.
2. Some neuron functions are multiargument (as F, G and L).
3. The outputs of neurons $\times$ and F in the second layer are coincident.

Thus, neural networks are not appropriate to reproduce this model.

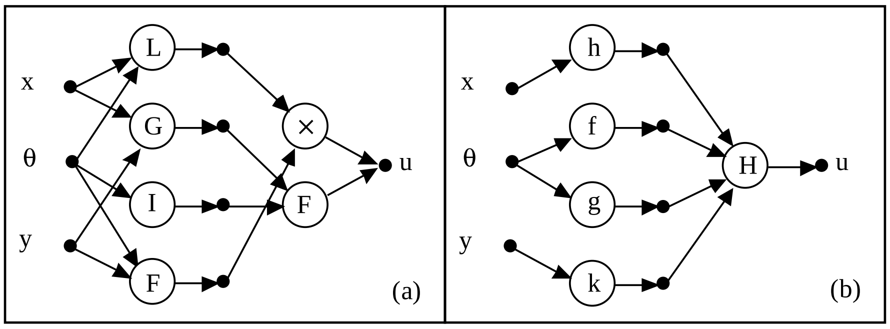


FIGURE 2.2. (a) Functional network associated with the Bayesian conjugate distributions problem, and (b) Equivalent simplified functional network.

Theorem 2.1 *Under some regularity conditions (see Castillo and Ruiz-Cobo (1992)), the general continuous solution of the functional equation*

$$F\left(G(x,y), z\right) = Q\left(M(x,z), N(y,z)\right), \tag{2.5}$$

where we assume $G \neq M, G \neq N, Q \neq M$ and $Q \neq N$, is

$$\begin{array}{rcl} F(x,y) & = & l[f(y)g^{-1}(x) + p(y) + q(y)], \\ G(x,y) & = & g[h(x) + k(y)], \\ Q(x,y) & = & l[m(x) + n(y)], \\ M(x,y) & = & m^{-1}[f(y)h(x) + p(y)], \\ N(x,y) & = & n^{-1}[f(y)k(x) + q(y)], \end{array} \tag{2.6}$$

where g, h, k, l, m and n are arbitrary strictly monotonic and continuously differentiable functions, and f, p and q are arbitrary continuously differentiable functions.

The two sides of (2.5) can be written as

$$l\{f(z)[h(x) + k(y)] + p(z) + q(z)\}$$

Functional equation (2.5) suggests a functional network as that in Figure 2.2(a) with the only difference being that the neuron $\times$ must be replaced by Q.

The functional equation (2.4) which has been obtained for the Bayesian conjugate case is a particular case of (2.5) with $Q(x, y) = xy$. A particular solution of functional equation (2.4) is (see Castillo and Ruiz-Cobo (1992)):

$$\begin{array}{rcl} F(\theta; y) & = & f(\theta)^{k(y)} g(\theta) \\ L(x; \theta) & = & f(\theta)^{h(x)} \\ G(x, y) & = & k^{-1}[h(x) + k(y)], \end{array} \tag{2.7}$$

showing that the two sides of (2.4) can be written as

$$f(\theta)^{h(x)+k(y)} g(\theta). \tag{2.8}$$

Thus, functional network in Figure 2.2(b) is equivalent to functional network in Figure 2.2(a), where the meaning of H is obvious from (2.8).

The functional network associated with Theorem 2.1 arise in many different problems. The reader is referred to Exercises 2.4 and 2.5 at the end of this chapter for two more examples.

2.3 Elements of a Functional Network

A functional network consists of the following elements (see Figure 2.1(a)):

1. *One layer of input storing units.* This layer contains the input data. Input units are represented by small black circles with their corresponding names (x, y and z in Figure 2.1(a)).

2. *One layer of output storing units.* This layer contains the output data. Output units are also represented by small black circles with their corresponding names (u in Figure 2.1(a)).

3. *One or several layers of processing units.* These units evaluate a set of input values, coming from the previous layer (of intermediate or input units) and delivers a set of output values to the next layer (of intermediate or output units). To this end, each neuron has associated a neuron function which can be multivariate, and can have as

many arguments as inputs. Each component (univariate) of a neural function is called a *functional cell.* Neurons are represented by ovals with the name of the corresponding function inside. For example, the functional network in Figure 2.1(a) has 9 neurons I, G, I, M, I, N, F, L and K.

4. *None, one or several layers of intermediate storing units.* These layers contain units that store intermediate information produced by neuron units. Intermediate units are represented by small black circles (there is one layer with 6 intermediate units in the functional network in Figure 2.1(a)). These layers allow forcing the outputs of processing units to be coincident.

5. *A set of directed links.* They connect units in the input or intermediate layers to neuron units, and neuron units to intermediate or output units. Connections are represented by arrows, indicating the information flow direction.

All these elements together form the *network architecture*, which defines the network topology and the associated functional capabilities of the network. The network architecture refers to the organization of the neurons and the connections involved and the functional capabilities refer to the specific functional form defined on the resulting network. Note that, as opposed to neural networks, in functional networks units are separated in two groups: storing and processing units.

The following definitions give a framework to work with functional networks.

Definition 2.1 (Functional Unit) *A functional unit (also called a neuron) over the set of nodes X, is a triplet (Y, F, Z), where Y and Z are disjoint non-empty subsets of X and $F : Y \to Z$ is a given function. The elements Y, Z and F are called the set of input nodes, the set of output nodes, and the processing function of the functional unit, respectively.*

Definition 2.2 (Functional Network) *A functional network is a pair (X, U), where X is a set of nodes and $U = \{(Y_i, F_i, Z_i)|i = 1, \ldots, n\}$ is a set of functional units over X, which satisfies the following condition: Every node $X_i \in X$ must be either an input or an output node of at least one functional unit in U.*

Figure 2.3 shows an example of a functional network. The set of nodes X parallels a printed circuit board (PCB) and the functional units parallels microcircuits or electronic components, thus giving an appealing and powerful interpretation to these elements.

This functional network contains five different functional units:

$$U_1 = (\{X_1, X_2\}, f_1, \{X_6\}),$$

$$\begin{aligned} U_2 &= (\{X_3, X_4\}, f_2, \{X_{10}\}), \\ U_3 &= (\{X_5, X_7, X_8\}, f_3, \{X_{10}, X_6\}), \\ U_4 &= (\{X_9, X_{10}\}, f_4, \{X_{12}, X_{14}\}), and \\ U_5 &= (\{X_{11}, X_{12}, X_{15}\}, f_5, \{X_{13}, X_{16}\}). \end{aligned} \tag{2.9}$$

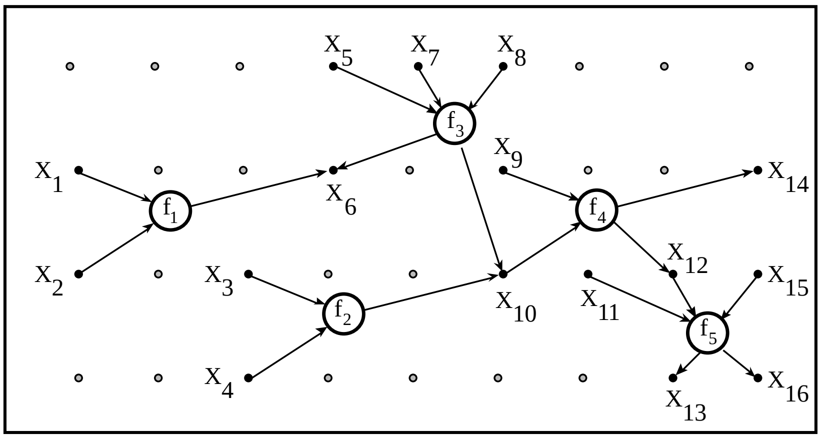


FIGURE 2.3. An example of functional network showing the set of nodes (printed circuit board) and five functional units (electronic components).

Definition 2.3 (Input Node of a Functional Network) *A node is said to be an input node of a functional network* (X, U)*, if it is the input node of at least one functional unit in* U *and is not the output of any functional unit in* U*.*

Definition 2.4 (Output Node of a Functional Network) *A node is said to be an output node of a functional network* (X, U)*, if it is the output node of at least one functional unit in* U *and is not the input of any functional unit in* U*.*

Definition 2.5 (Intermediate Node of a Functional Network) *A node is said to be an intermediate node of a functional network* (X, U)*, if it is the input node of at least one functional unit in* U *and, at the same time, is the output node of at least one functional unit in* U*.*

For example, in Figure 2.3 X_1 and X_2 are input nodes, X_6 and X_{10} are intermediate nodes, and X_{14} and X_{16} are output nodes of the functional network.

Remark 2.1 *Definition 2.2 together with Definitions 2.3, 2.4 and 2.5 implies that* X *can be partitioned in three subsets*

$$\{X_{in}, X_{inter}, X_{out}\},$$

where X_{inter} can be the empty set. When this happens, the functional network is called a one-layer network.

2.4 Differences Between Neural and Functional Networks

The characteristics and key features of functional networks, as compared with standard neural networks are (see Figure 2.1(a)):

1. In selecting the topology of functional networks, the required information can be derived from data, from domain knowledge or from different combinations of the two. In the case of standard neural networks, only the data is used. This implies that, in addition to the data information, other properties of the function being modeled by the functional network can be used for selecting its topology (associativity, commutativity, invariance, etc.). This information is available in some practical cases.

2. Unlike standard neural networks, where the neuron functions are assumed to be fixed and known and only the weights are learned, in functional networks the functions are learned during the structural learning (which obtains simplified network and functional structures) and estimated during the parametric learning (which consists of obtaining the optimal neuron function from a given family).

3. Arbitrary neural functions can be assumed for each neuron (see, for example, neurons F, G, I, K, L, M and N in Figure 2.1(a)), while in neural networks they are fixed *sigmoidal* functions.

4. No weights are used, since they can be incorporated into the neural functions.

5. The neural functions are allowed to be truly multiargument, such as neural functions F, G, K, L, M and N in Figure 2.1(a). However, as we shall see, in many cases they can be equivalently replaced by functions of single variables.

   Note that in standard neural networks the neural sigmoidal functions are of a single argument though this is a linear combination of all inputs (pseudo-multiargument functions).

6. Unlike neural networks, intermediate or output units can be connected (linked) to several storing units, say m units, indicating that the associated values must be equal. Each of this common connections represents a functional constrain in the model and allows writing the value of these output units in different forms (one per different link).

This leads to a system of $m-1$ functional equations. By solving this system the initial neuron functions can be simplified, for example, reducing the number of arguments. In Figure 2.1(a), the outputs of $m=3$ neurons F, L and K are connected. Thus, leading to a system of two functional equations (the last two equations in (2.1)).

Intermediate layers of units are introduced in functional network architectures to allow several neuron outputs to be connected to the same units (this is not possible in neural networks).

7. Functional networks are extensions of neural networks.

   It is important to point out that neural networks are special cases of functional networks. For example, in Figure 2.4 a neural network and its equivalent functional network are shown. Note that weights are subsumed by the neural functions.

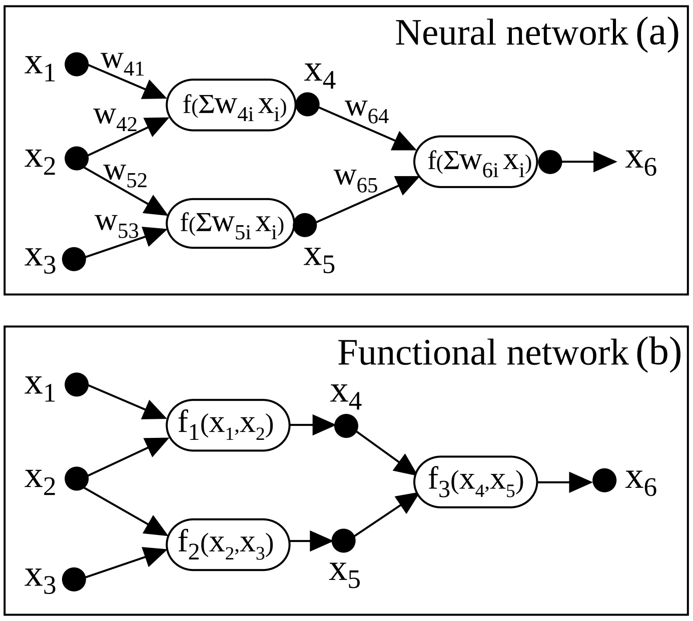


FIGURE 2.4. (a) Neural network, and (b) its equivalent functional network.

Standard neural networks can be modified to reduce these differences; for example, by constraining the weights associated with the output links. This would result in a reduced set of connection weights to be determined. Some of the other features of functional nets, including invariance properties, can also be realized in neural networks (see Abu-Mostafa (1990) or Suddarth and Holden (1991)).

It can be argued that the proposed functional networks are based on functional equations and use parametric modeling methods; in other words, that they require domain knowledge for deriving the functional equations and make assumptions about the form the unknown functions should take, and that standard backpropagation neural networks do not have these limitations and provide a non-parametric approach to modeling. However, this is not true. In fact, neural networks have parameters (weights), and functional networks can also be used as black boxes, as neural networks.

2.5 Working With Functional Networks

Working with functional networks requires the following steps:

Step 1 (Statement of the problem): Understanding of the problem to be solved. This is a crucial step.

Step 2 (Initial topology): Based on the knowledge of the problem, the topology of the initial functional network is selected. This selection is not as in neural networks, where several topologies are used until errors are small and balanced with the number of degrees of freedom. The selection of the topology of a functional network is normally based on properties, which usually lead to a clear and single network structure (problem driven design).

Step 3 (Simplification or structural learning): The initial functional network is simplified using functional equations. Given a functional network, an interesting problem consists of determining whether or not there exists another functional network giving the same output for every given input. This leads to the concept of equivalent functional networks. Two functional networks are said to be equivalent if they have the same input and output layers and they give the same output for any given input. The practical importance of the concept of equivalent functional networks is that we can define equivalent classes of functional networks, that is, sets of equivalent functional networks, and then choose the simplest in each class to be used in applications. Functional equations are the main tool for simplifying functional networks (for a general introduction to functional equations see Aczél (1966) and Castillo and Ruiz-Cobo (1992)).

Step 4 (Uniqueness of representation): Before learning a functional network we need to be sure that there is a unique representation of it. In other words, for a given topology (structure), in some cases, there are several neuron functions leading to exactly the same output for any input. To avoid estimation problems we need to know what conditions must hold for uniqueness.

Step 5 (Data collection): For the learning to be possible, some information is required. In this step the data is collected.

Step 6 (Parametric learning): The neural functions are estimated (learned) based on the given data. This is done by considering linear combinations of appropriate functional families and using some minimization method to obtain the optimal coefficients.

There are two different types of learning methods:

1. **The Linear Method**: It is called linear because the associated optimization function leads to a system of linear equations in the parameter estimates. In this case, a single minimum exists and the learning process reduces to solving a system of linear equations.

2. **The Non-Linear Method**: It leads to a function which is nonlinear in the parameters. In this case there may exist multiple minima and the optimization process can be carried out by considering some standard gradient descendent method.

Step 7 (Model validation): The test for quality and/or the cross validation of the model is performed. Checking the obtained error is important to see whether or not the selected family of approximating functions are adequate. A cross validation of the model, with an alternative set of data, is also important. This allows deciding on the existence of the overfitting problem.

Step 8 (Use of the model): If the validation process is satisfactory, the model is ready to be used.

2.6 An Introductory Example

In this section we illustrate the differences between functional and neural network by using a simple example. Suppose that we have the set of data points (x_1, x_2, x_3) shown in Table 2.1, obtained from a function $x_3 = F(x_1, x_2)$. Suppose also that we do not have any information about the form of the function, but we know that it is associative, i.e., the F function satisfies

$$F(F(x_1, x_2), x_3) = F(x_1, F(x_2, x_3)). \tag{2.10}$$

One of the possibilities of obtaining an approximate model for this set of data consists of using a neural network model with two inputs, x_1 and x_2, one output x_3, and a hidden layer. With the aim of checking both the quality and the prediction quality of the resulting models, the data is separated in two groups: the first 30 points will be used to train the network, and the last 30 points to perform a cross validation.

x_1	x_2	x_3	x_1	x_2	x_3	x_1	x_2	x_3
0.413	0.186	0.453	0.552	0.645	0.849	0.393	0.967	1.044
0.514	0.492	0.711	0.211	0.260	0.336	0.517	0.836	0.983
0.515	0.468	0.696	0.426	0.746	0.859	0.793	0.747	1.089
0.682	0.894	1.125	0.224	0.221	0.315	0.720	0.120	0.730
0.811	0.034	0.812	0.168	0.475	0.504	0.418	0.067	0.423
0.653	0.983	1.180	0.206	0.807	0.833	0.137	0.147	0.201
0.690	0.339	0.769	0.710	0.404	0.816	0.897	0.593	1.075
0.028	0.507	0.508	0.672	0.372	0.768	0.307	0.387	0.494
0.861	0.338	0.925	0.140	0.912	0.923	0.443	0.270	0.519
0.486	0.929	1.049	0.238	0.463	0.521	0.350	0.782	0.857
0.548	0.124	0.561	0.639	0.381	0.745	0.651	0.531	0.840
0.612	0.873	1.067	0.979	0.159	0.992	0.305	0.486	0.574
0.118	0.821	0.829	0.164	0.574	0.597	0.675	0.551	0.871
0.679	0.645	0.936	0.437	0.088	0.446	0.329	0.862	0.923
0.889	0.964	1.311	0.690	0.481	0.841	0.238	0.433	0.494
0.077	0.607	0.612	0.259	0.275	0.377	0.773	0.121	0.782
0.141	0.454	0.475	0.608	0.546	0.818	0.466	0.903	1.016
0.930	0.902	1.295	0.029	0.815	0.816	0.600	0.039	0.602
0.140	0.851	0.862	0.911	0.558	1.068	0.902	0.417	0.994
0.833	0.951	1.264	0.643	0.143	0.659	0.060	0.830	0.832

TABLE 2.1. Set of simulated data for the associative operation.

m	parameters	RMSE training	RMSE testing
2	9	0.0551	0.0712
4	17	0.0019	0.0042
6	25	0.0018	0.0031

TABLE 2.2. Errors for a multilayer perceptron with m hidden units.

Several neural networks containing different numbers of hidden units, say m neurons, have been considered performing $10,000$ iterations of the back-propagation method for each of the cases. Different initial weight configurations have been considered and the best results have been chosen. Both the training and testing RMSEs (root mean squared errors) are shown in Table 2.2.

Let us solve this problem by using functional networks, following the steps indicated in the previous section.

Step 1 (Statement of the problem): We wish to reproduce the above associative operation F between two real numbers, i.e., the F function satisfies (2.10). Note that $F(x_1, x_2)$ summarizes the contribution of x_1 and x_2

to $F(F(x_1, x_2), x_3)$. In fact associativity means that we can operate consecutive pairs of operands in any order.

Step 2 (Initial topology): Equation (2.10) suggests the initial network topology shown in Figure 2.5(a).

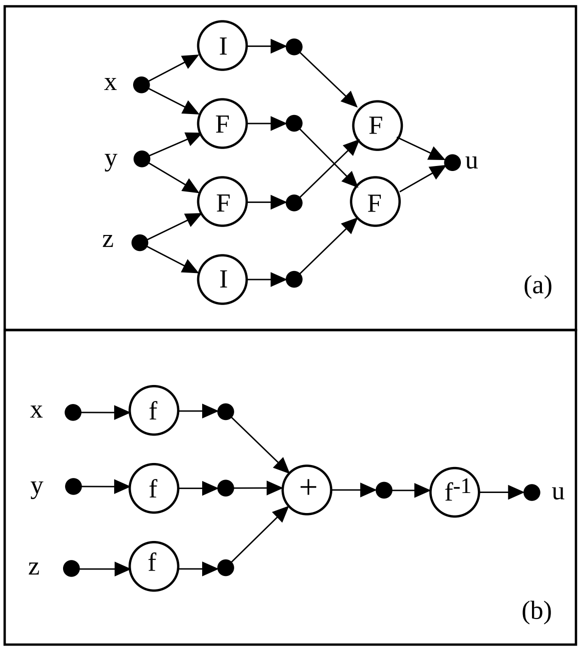


FIGURE 2.5. Illustration of the associativity functional network. (a) Initial network, and (b) Equivalent simplified network.

Step 3 (Simplification or structural learning): Initially, it seems that a two-argument function F has to be learned. However, the functional equation (2.10) puts strong constraints on it. In fact, the general solution of the functional equation (2.10) is given by the following theorem (see Aczél, 1966, or Castillo and Ruiz-Cobo, 1992).

Theorem 2.2 (Associativity Equation) *The general solution continuous and invertible in both variables on a real rectangle of the functional equation (2.10) is*

$$F(x_1, x_2) = f^{-1}[f(x_1) + f(x_2)], \tag{2.11}$$

where $f(x)$ is an arbitrary continuous and strictly monotonic function, which can be replaced only by $cf(x)$, where c is an arbitrary constant. ■

The two most common examples of associative operations are the sum and the product. For the sum we have:

$$\begin{aligned} f(x) &= x \\ F(x_1, x_2) &= x_1 + x_2 \end{aligned}$$

and for the product:

$$\begin{aligned} f(x) &= \log(x) \\ F(x_1, x_2) &= \exp[\log(x_1) + \log(x_2)] = x_1 x_2 \end{aligned}$$

Replacing (2.11) in (2.10), we can see that the two sides of (2.10) can be written as

$$f^{-1}[f(x_1) + f(x_2) + f(x_3)], \tag{2.12}$$

and then, the functional network in Figure 2.5(a) is equivalent to the functional network in Figure 2.5(b), where only a one-argument function f need to be learned.

Figure 2.6 shows a network able to perfectly reproduce any associative operation (see Equation (2.11)).

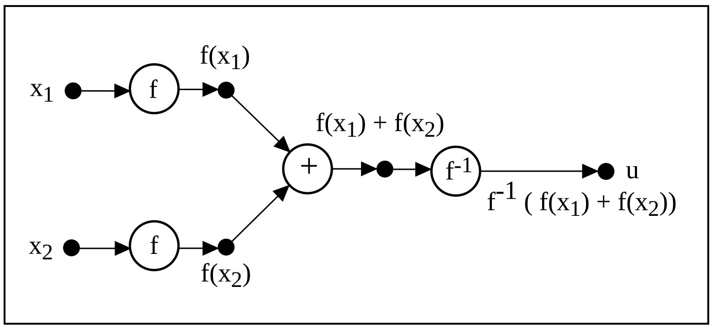


FIGURE 2.6. Graphical illustration of the network associated with $F(x_1, x_2) = u$, where F corresponds to an associative operation.

Two important conclusions can be derived from this theorem:

1. No other functional forms for F satisfy equation (2.10). So, no other neurons can be replaced by neurons F.

2. The initial two-dimensional function F is completely determined by means of a unidimensional function f. Thus, the functional equation (2.10) reduces the initial degrees of freedom of $F(\cdot, \cdot)$ from a two-argument function to a single-argument function f.

Step 4 (Uniqueness of representation): The functional structure of the solution is (2.2), where f is determined up to a constant. In other words, the constant c is not identifiable from the functional equation alone, i.e., one extra condition is required to obtain c. However, no matter which value of c is used, Expressions (2.2) and (2.12) give the same function, i.e., there are many different functional networks which are equivalent. Thus, a concrete value of c is not needed.

Step 5 (Data collection): To learn this associative operation in the interval $(0, 5)$, we can take pairs of numbers in that interval and their operated values as triplets

$$\{(x_{1j}, x_{2j}, x_{3j}) | x_{3j} = F(x_{1j}, x_{2j}) = x_{1j} \oplus x_{2j}; j = 1, \ldots, n\}.$$

Note that, for convenience, we have used the notation x_1, x_2 and x_3 for x, y and z, respectively.

We can do this deterministically or we can simulate a set of triplets. Assume that we simulate 60 of these triplets and obtain the values in Table 2.1.

Step 6 (Learning): From (2.2) we get

$$u = F(x_1, x_2) \Leftrightarrow f(u) = f(x_1) + f(x_2), \tag{2.13}$$

an interesting relation to be exploited for learning $f(x)$.

Learning this network is equivalent to learning the function $f(x)$. To this end, we can approximate $f(x)$ by

$$\hat{f}(x) = \sum_{i=1}^{m} a_i \phi_i(x), \tag{2.14}$$

where the $\{\phi_i(x); i = 1, \ldots, m\}$ is a set of given linearly independent functions, capable of approximating $f(x)$ to the desired accuracy.

To estimate the coefficients $\{a_i; i = 1, \ldots, m\}$, we use the collected data in the form of triplets (x_{1j}, x_{2j}, x_{3j}). According to (2.13) we must have

$$f(x_{3j}) = f(x_{1j}) + f(x_{2j}); j = 1, \ldots, n, \tag{2.15}$$

thus, the error of the approximation can be measured by

$$e_j = \hat{f}(x_{1j}) + \hat{f}(x_{2j}) - \hat{f}(x_{3j}); j = 1, \ldots, n. \tag{2.16}$$

To estimate the coefficients $\{a_i; i = 1, \ldots, m\}$, we minimize the sum of squared errors

$$Q = \sum_{j=1}^{n} e_j^2 = \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_i \left[\phi_i(x_{1j}) + \phi_i(x_{2j}) - \phi_i(x_{3j}) \right] \right)^2 \tag{2.17}$$

subject to

$$f(x_0) \equiv \sum_{i=1}^{m} a_i \phi_i(x_0) = \alpha, \tag{2.18}$$

where α is an arbitrary but given real constant, which is necessary to identify the otherwise unidentifiable constant c in Theorem 2.2.

Thus, using the Lagrange multipliers technique we build the auxiliary function

$$Q_\lambda = \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_i b_{ij} \right)^2 + \lambda \left(\sum_{i=1}^{m} a_i \phi_i(x_0) - \alpha \right), \tag{2.19}$$

where

$$b_{ij} = \phi_i(x_{1j}) + \phi_i(x_{2j}) - \phi_i(x_{3j}). \tag{2.20}$$

The minimum corresponds to

$$\begin{aligned} \frac{\partial Q_\lambda}{\partial a_r} &= 2\sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_i b_{ij} \right) b_{rj} + \lambda \phi_r(x_0) = 0; \ r = 1, \ldots, m, \\ \frac{\partial Q_\lambda}{\partial \lambda} &= \sum_{i=1}^{m} a_i \phi_i(x_0) - \alpha = 0, \end{aligned} \tag{2.21}$$

which is a linear system of $m+1$ equations and $m+1$ unknowns having a unique solution, assuming the set of functions $\{\phi_i; i = 1, \ldots, m\}$ to be linearly independent. In matrix form, it can be written as

$$\begin{pmatrix} \mathbf{B}\mathbf{B}^T & \boldsymbol{\phi}_0 \\ \boldsymbol{\phi}_0^T & 0 \end{pmatrix} \begin{pmatrix} \mathbf{a}^\mathbf{T} \\ \lambda \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \alpha \end{pmatrix}, \tag{2.22}$$

where $\mathbf{B}$ is the matrix of coefficients b_{ij}, $\mathbf{B}^T$ is the transpose of $\mathbf{B}$, $\boldsymbol{\phi}_0 = (\phi_1(x_0), \ldots, \phi_m(x_0))$ and $\mathbf{a} = (a_1, \ldots, a_m)$.

If now we take, for example, a polynomial family

$$\{\phi_i(x); i = 1, \ldots, m\} = \{1, x, x^2\},$$

with $m = 3$, and solve the linear system of equations (2.22), we obtain

$$(a_1, a_2, a_3, \lambda) = (0, 0, 1, 0)$$

and zero error, indicating that the associative operator corresponding to the above data is $F(x_1, x_2) = f^{-1}(f(x_1) + f(x_2)) = \sqrt{x_1^2 + x_2^2}$.

If now we take, for example, a Fourier family

$$\{1, sin(x), \ldots, sin(m\,x), cos(x), \ldots, cos(m\,x),\},$$

for different values of m, then we get different approximate models for the function. For example, for $m = 1$, we get the model $\hat{f}(x) = 2.407 - 2.389 cos(x) - 0.138 sin(x)$.

Step 7 (Model validation): To check the quality of the approximation, we have calculated the training and testing RMSE errors for several models. The results are shown in Table 2.3. Comparing this table with the one corresponding to neural networks (Table 2.2) we can see how functional networks outperform neural networks in this example.

m	parameters	RMSE train	RMSE test
1	3	$6.9\,10^{-3}$	$9.4\,10^{-3}$
2	5	$1.2\,10^{-4}$	$3.6\,10^{-4}$
3	7	$2.7\,10^{-6}$	$1.1\,10^{-5}$

TABLE 2.3. RMSE for the approximated sums using $\{1, sin(x), \ldots, sin(m\,x), cos(x), \ldots, cos(m\,x), \}$ with $m = 1, 2, 3$.

Step 8 (Use of the model): Once the model has been tested satisfactorily, it is ready to be used for the associativity operation.

Exercises

2.1 Solve the functional equation

$$F[G(x, y), z] = K[x, N(y, z)],$$

and draw its associated and the resulting simplified functional networks.

Hint: Use the solution in (2.2).

2.2 Design a functional network to obtain the most general surface such that its intersections with planes parallel to the coordinate planes ZY and ZX are linear combinations of functions in the sets $\{\phi_1(x), \phi_2(x), \phi_3(x)\}$ and $\{\psi_1(y), \psi_2(y), \psi_3(y)\}$, respectively.

2.3 Show that the standard general solution

$$F(x, y) = f^{-1}[f(x) + f(y)]$$

of the associativity functional equation

$$F(F(x, y), z) = F(x, F(y, z))$$

can also be written as

$$F(x, y) = f^{-1}[f(x) \oplus f(y)],$$

where $\oplus$ is any associative operation.

2.4 If $F_X(x)$ and $F_y(y)$ are the cumulative distribution functions of two independent random variables X and Y, the cumulative distribution function $F_Z(z)$ of $Z = max(X, Y)$ is

$$F_Z(z) = F_X(z)F_Y(y).$$

(a) Write the functional equation for X and Z to belong to the same uniparametric family, and draw the corresponding functional network.

(b) Write the functional equation for X, Y and Z to belong to the same uniparametric family, i.e., for the uniparametric family of distributions being closed with respect to maximum operations, and draw the corresponding functional network.

2.5 We say that a family of random variables is reproductive under convolution if the sum of independent random variables of the family belongs to the family, that is, if the family is closed under sums. We know that the characteristic function $\phi_Z(t)$ of the sum $Z = X + Y$ of two independent random variables is the product of their characteristic functions $\phi_X(t)$ and $\phi_Y(t)$ of the summands X and Y:

$$\phi_Z(t) = \phi_X(t)\phi_Y(t),$$

Thus, reproductivity of parametric families can be written as a functional equation. Write the functional equation for the case of uniparametric families. Draw the functional network associated with this equation. Simplify the network by solving the resulting functional equation as a particular case of (2.5).

Hint: Decompose the resulting functional equation in a complex function in two functional equations in two real functions.

2.6 Using the general solution (2.2) of the functional equation (2.1), solve the system of functional equations

$$F[G(x, y), z) = F[x, N(y, z)] = F[y, M[x, z)].$$