CHAPTER

NEURAL NETWORKS AND DEEP LEARNING

18

CHAPTER OUTLINE

Machine Learning. http://dx.doi.org/10.1016/B978-0-12-801522-3.00018-5

18.1 INTRODUCTION

Neural networks have a long history that goes back to the first attempts to understand how the human (and more generally, the mammal) brain works and how what we call intelligence is formed.

From a physiological point of view, one can trace the beginning of the field back to the work of Santiago Ramon y Cajal, [68], who discovered that the basic building element of the brain is the *neuron*. The brain comprises approximately 60 to 100 billions neurons; that is, a number of the same order as the number of stars in our galaxy! Each neuron is connected with other neurons via elementary structural and functional units/links, known as *synapses*. It is estimated that there are 50 to 100 trillions of synapses. These links mediate information between connected neurons. The most common type of synapses are the chemical ones, which convert electric pulses, produced by a neuron, to a chemical signal and then back to an electrical one. Depending on the input pulse(s), a synapse is either *activated* or *inhibited*. Via these links, each neuron is connected to other neurons and this happens in a hierarchically structured way, in a layer-wise fashion.

Santiago Ramon y Cajal (1852–1934) was a Spanish pathologist, histologist, neuroscientist, and Nobel laureate. His many pioneering investigations of the microscopic structure of the brain have established him as the father of modern neuroscience.

A milestone from the learning theory's point of view occurred in 1943, when Warren McCulloch and Walter Pitts, [59], developed a computational model for the basic neuron. Moreover, they provided results that tie neurophysiology with mathematical logic. They showed that given a sufficient number of neurons and adjusting appropriately the synaptic links, each one represented by a weight, one can compute, in principle, any computable function. As a matter of fact, it is generally accepted that this is the paper that gave birth to the fields of neural networks and artificial intelligence.

Warren McCulloch (1898–1969) was an American psychiatrist and neuroanatomist who spent many years studying the representation of an event in the neural system. Walter Pitts (1923–1969) was an American logician who worked in the field of cognitive psychology. He was a mathematical prodigy and he taught himself logic and mathematics. At the age of 12, he read *Principia Mathematica* by Alfred North Whitehead and Bertrand Russell and he wrote a letter to Russell commenting on certain parts of the book. He worked with a number of great mathematicians and logicians including Wiener, Householder, and Carnap. When he met McCulloch at the University of Chicago, he was familiar with the work of Leibnitz on computing, which inspired them to study whether the nervous system could be considered to be a type of universal computing device. This gave birth to their 1943 paper, mentioned in the reference before.

Frank Rosenblatt, [73, 74], borrowed the idea of a neuron model, as suggested by McCulloch and Pitts, to build a true learning machine which learns from a set of training data. In the most basic version of operation, he used a single neuron and adopted a rule that can learn to separate data, which belong to two linearly separable classes. That is, he built a pattern recognition system. He called the basic neuron a *perceptron* and developed a rule/algorithm, the *perceptron algorithm*, for the respective training. The perceptron will be the kick-off point for our tour in this chapter.

Frank Rosenblatt (1928–1971) was educated at Cornell, where he obtained his PhD in 1956. In 1959, he took over as director of Cornell's Cognitive Systems Research Program and also as a lecturer in the psychology department. He used an IBM 704 computer to simulate his perceptron and later built a special-purpose hardware, which realized the perceptron learning rule.

Neural networks are learning machines, comprising a large number of neurons, which are connected in a layered fashion. Learning is achieved by adjusting the unknown synaptic weights to minimize a preselected cost function. It took almost 25 years, after the pioneering work of Rosenblatt, for neural networks to find their widespread use in machine learning. This is the time period needed for the basic McCulloch Pitts's model of a neuron to be generalized and lead to an algorithm for training such networks. A breakthrough came under the name *backpropagation algorithm*, which was developed for training neural networks based on a set of input-output training samples. Backpropagation is also treated in detail in this chapter.

It is interesting to note that neural networks dominated the field of machine learning for almost a decade, from 1986 until the middle of 1990s. Then they were superseded, to a large extent, by the support vector machines, which established their reign until very recently. At the time the book is being compiled, there is an aggressive resurgence in the interest on neural networks, in the context of *deep learning*. Interestingly enough, there is one name that is associated with the revival of interest on neural networks, both in the mid-1980s and now; this is the name of Geoffrey Hinton [34, 75]. Deep learning refers to learning networks with many layers of neurons, and this topic is treated at the end of this chapter.

18.2 THE PERCEPTRON

Our starting point is the simple problem of a *linearly separable* two-class (ω_1, ω_2) classification task. In other words, we are given a set of training samples, $(y_n, \boldsymbol{x}_n)$, $n = 1, 2, \ldots, N$, with $y_n \in \{-1, +1\}$, $\boldsymbol{x}_n \in \mathbb{R}^l$ and it is assumed that there is a hyperplane,

$$\boldsymbol{\theta}_*^T \boldsymbol{x} = 0$$

such that,

$$\begin{aligned} \boldsymbol{\theta}_*^T \boldsymbol{x} &> 0, \quad \text{if } \boldsymbol{x} \in \omega_1, \\ \boldsymbol{\theta}_*^T \boldsymbol{x} &< 0, \quad \text{if } \boldsymbol{x} \in \omega_2. \end{aligned}$$

In other words, such a hyperplane classifies correctly *all* the points in the training set. For notational simplification, the bias term of the hyperplane has been absorbed in $\boldsymbol{\theta}_*$ after extending the dimensionality of the problem by one, as it has been explained in Chapter 3 and used in various parts of this book.

The goal now becomes that of developing an algorithm that iteratively computes a hyperplane that classifies correctly all the patterns from both classes. To this end, a cost function is adopted.

The Perceptron Cost. Let the available estimate at the current iteration step of the unknown parameters be $\boldsymbol{\theta}$. Then, there are two possibilities. The first one is that all points are classified correctly; this means that a solution has been obtained. The other alternative is that $\boldsymbol{\theta}$ classifies correctly some of the points and the rest are misclassified. Let $\mathcal{Y}$ be the set of all misclassified samples. The perceptron cost is defined as

$$\boxed{J(\boldsymbol{\theta}) = -\sum_{n:\boldsymbol{x}_n \in \mathcal{Y}} y_n \boldsymbol{\theta}^T \boldsymbol{x}_n : \quad \text{Perceptron Cost,}} \tag{18.1}$$

where,

$$y_n = \begin{cases} +1, & \text{if } \boldsymbol{x} \in \omega_1, \\ -1, & \text{if } \boldsymbol{x} \in \omega_2. \end{cases} \tag{18.2}$$

Observe that the cost function is nonnegative. Indeed, because the sum is over the misclassified points, if $\boldsymbol{x}_n \in \omega_1$ (ω_2) then $\boldsymbol{\theta}^T \boldsymbol{x}_n < (>) \, 0$ rendering the product $-y_n \boldsymbol{\theta}^T \boldsymbol{x}_n > 0$. The cost function becomes zero, if there are no misclassified points, that is, $\mathcal{Y} = \emptyset$, which corresponds to a solution.

The perceptron cost function is not differentiable at all points. It is a *continuous piece-wise linear* function. Indeed, let us write it in a slightly different way,

$$J(\boldsymbol{\theta}) = \left(- \sum_{n: \boldsymbol{x}_n \in \mathcal{Y}} y_n \boldsymbol{x}_n^T \right) \boldsymbol{\theta},$$

This is a linear function with respect to $\boldsymbol{\theta}$, as long as the number of misclassified points remains the same. However, as one slowly changes the value of $\boldsymbol{\theta}$, which corresponds to a change of the position of the respective hyperplane, there will be a point where the number of misclassified samples in $\mathcal{Y}$ suddenly changes; this is the time, where a sample in the training set changes its relative position with respect to the (moving) hyperplane and as a consequence the set $\mathcal{Y}$ is modified. After this change, $J(\boldsymbol{\theta})$ will correspond to a new linear function.

The Perceptron Algorithm. It can be shown, for example, [61, 74], that, starting from an arbitrary point, $\boldsymbol{\theta}^{(0)}$, the following iterative update,

$$\boxed{\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} + \mu_i \sum_{n: \boldsymbol{x}_n \in \mathcal{Y}} y_n \boldsymbol{x}_n : \quad \text{The Perceptron Rule,}} \tag{18.3}$$

converges after a *finite number of steps*. The parameter μ_i is the user-defined step size, judicially chosen to guarantee convergence. Note that this is the same algorithm as the one derived in Section 8.10.2, for minimizing the hinge loss function via the notion of subgradient.

Besides the previous scheme, another version of the algorithm considers one sample per iteration in a cyclic fashion, until the algorithm converges. Let us denote by $y_{(i)}$, $\boldsymbol{x}_{(i)}$, $(i) \in \{1, 2, \ldots, N\}$, the training pair that is presented in the algorithm at the ith iteration step.[1] Then, the update iteration becomes

$$\boxed{\boldsymbol{\theta}^{(i)} = \begin{cases} \boldsymbol{\theta}^{(i-1)} + \mu_i y_{(i)} \boldsymbol{x}_{(i)}, & \text{if } \boldsymbol{x}_{(i)} \text{ is misclassified by } \boldsymbol{\theta}^{(i-1)}, \\ \boldsymbol{\theta}^{(i-1)}, & \text{otherwise.} \end{cases}} \tag{18.4}$$

In other words, starting from an initial estimate, usually taken to be equal to zero, $\boldsymbol{\theta}^{(0)} = \mathbf{0}$, we test each one of the samples, $\boldsymbol{x}_n$, $n = 1, 2, \ldots, N$. Every time a sample is misclassified, action is taken for a correction. Otherwise no action is required. Once all samples have been considered, we say that one *epoch* has been completed. If no convergence has been attained, all samples are reconsidered in a

[1] The symbol (i) has been adopted to denote the time index of the samples, instead of i, because we do not know which point will be presented to the algorithm at the ith iteration. Recall that each training point is considered many times, till convergence is achieved.

second epoch and so on. This is known as *pattern-by-pattern* or *online* mode of operation. However, note that, in contrast to how we have used the term "online" in previous chapters, here we mean that the total number of data samples is fixed and the algorithm considers them in a *cyclic fashion*, epoch after epoch.

After a successive *finite* number of epochs, the algorithm is guaranteed to converge. Note that for convergence, the sequence μ_i must be appropriately chosen. This is pretty familiar to us by now. However for the case of the perceptron algorithm, convergence is still guaranteed even if μ_i is a positive constant, $\mu_i = \mu > 0$, usually taken to be equal to one (Problem 18.1).

The formulation in (18.4) brings the perceptron algorithm under the umbrella of the so-called *reward-punishment* philosophy of learning. If the current estimate succeeds in predicting the class of the respective pattern, no action is taken. Otherwise, the algorithm is punished to perform an update.

Figure 18.1 provides a geometric interpretation of the perceptron rule. Assume that sample $\boldsymbol{x}$ is misclassified by the hyperplane, $\boldsymbol{\theta}^{(i-1)}$. As we know from geometry, $\boldsymbol{\theta}^{(i-1)}$ corresponds to a vector that is perpendicular to the hyperplane that defines; see also Figure 11.15 in Section 11.10.1. Because $\boldsymbol{x}$ lies in the $(-)$ side of the hyperplane and it is misclassified, it belongs to class ω_1. Hence, assuming $\mu = 1$, the applied correction by the algorithm is

$$\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} + \boldsymbol{x},$$

and its effect is to turn the hyperplane to the direction toward $\boldsymbol{x}$ to place it in the $(+)$ side of the new hyperplane, which is defined by the updated estimate $\boldsymbol{\theta}^{(i)}$. The perceptron algorithm in its pattern-by-pattern mode of operation is summarized in Algorithm 18.1.

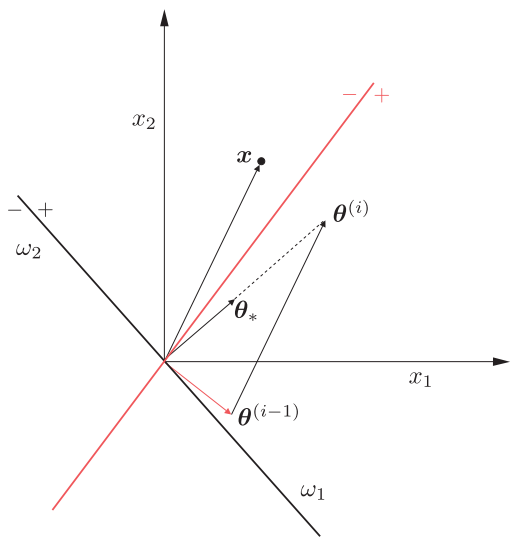

FIGURE 18.1

Pattern $\boldsymbol{x}$ is misclassified by the red line. The action of the perceptron rule is to turn the hyperplane toward the point $\boldsymbol{x}$, in an attempt to include it in the correct side of the new hyperplane and classify it correctly.

Algorithm 18.1 (The online perceptron algorithm).

- Initialization
  - $\boldsymbol{\theta}^{(0)} = \mathbf{0}$.
  - Select μ; usually it is set equal to one.
  - $i = 0$.
- **Repeat**; Each iteration corresponds to an epoch.
  - counter = 0; Counts the number of updates per epoch.
  - **For** $n = 1, 2, \ldots, N$, **Do**; For each epoch, all samples are presented.
    - **If** $(y_n \boldsymbol{x}_n^T \boldsymbol{\theta}^{(i-1)} \leq 0)$ **Then**
      - $i = i + 1$
      - $\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} + \mu y_n \boldsymbol{x}_n$
      - counter=counter+1
  - **End For**
- **Until** counter=0

Once the perceptron algorithm has run and converged, we have the weights, θ_i, $i = 1, 2, \ldots, l$, of the synapses of the associated neuron/perceptron as well as the bias term θ_0. These can now be used to classify unknown patterns. Figure 18.2a shows the corresponding architecture of the basic neuron element. The features, x_i, $i = 1, 2, \ldots, l$, are applied to the input nodes. In turn, each feature is multiplied by the respective synapse (weight), and then the bias term is added on their linear combination. The outcome of this operation then goes through a nonlinear function, f, known as the *activation* function. Depending on the form of the nonlinearity, different types of neurons occur. In the more classical one, known as the McCulloch-Pitts neuron, the activation function is the Heaviside one, that is,

$$f(z) = \begin{cases} 1 & \text{if } z > 0, \\ 0 & \text{if } z \leq 0. \end{cases} \tag{18.5}$$

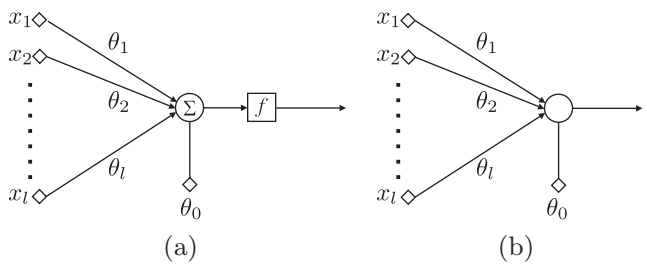

FIGURE 18.2

(a) In the basic neuron/perceptron architecture the input features are applied to the input nodes and are weighted by the respective weights of the synapses. The bias term is then added on their linear combination and the result is pushed through the nonlinearity. In the McCulloch-Pitts neuron, the output fires a 1 for patterns in class ω_1 or a zero for the other class. (b) The summation and nonlinear operation are merged together for graphical simplicity.

Usually, the summation operation and the nonlinearity are merged to form a node and the architecture in Figure 18.2b occurs.

Remarks 18.1.

- *ADALINE*: Soon after Rosenblatt proposed the perceptron, Widrow and Hopf proposed the *adaptive line element* (ADALINE), which is a linear version of the perceptron [98]. That is, during training the nonlinearity of the activation function is not involved. The resulting algorithm is the LMS algorithm, treated in detail in Chapter 5. It is interesting to note that the LMS was readily adopted and widely used for online learning within the signal processing and communications communities.

18.2.1 THE KERNEL PERCEPTRON ALGORITHM

In Chapter 11,[2] we discussed the kernelization of various linear algorithms to exploit Cover's theorem and solve a linear task in a reproducing kernel Hilbert space (RKHS), although the original problem is not (linearly) separable in the original space. The perceptron algorithm is not an exception and a kernelized version can be developed. Our starting point is the pattern-by-pattern version of the perceptron algorithm. Every time a pattern, $\boldsymbol{x}_{(i)}$, is misclassified, a correction to the parameter vector is performed as in (18.4). The difference now is that the place of $\boldsymbol{x}_{(i)}$ is taken by the image $\boldsymbol{\phi}(\boldsymbol{x}_{(i)})$, where $\boldsymbol{\phi}$ denotes the (implicit) mapping into the respective RKHS, as explained in Chapter 11. Let us introduce a new variable, a_n, which counts how many times the corresponding feature vector, $\boldsymbol{x}_n$, has participated in the correction update. Then, after convergence (we assume that after the mapping in the RKH space, the classes have become linearly separable and the algorithm is guaranteed to converge) the parameter,[3] $\boldsymbol{\theta}$, as well as the bias term, can be written as

$$\boldsymbol{\theta} = \sum_{n=1}^{N} a_n y_n \boldsymbol{\phi}(\boldsymbol{x}_n), \tag{18.6}$$

$$\theta_0 = \sum_{n=1}^{n} a_n y_n, \tag{18.7}$$

where we have considered the bias term separately and the dimension of the input vectors has not been extended. Then, given an unknown pattern, $\boldsymbol{x}$, classification is performed according to the sign of

$$f(\boldsymbol{x}) := < \boldsymbol{\theta}, \phi(\boldsymbol{x}) > + \theta_0 = \sum_{n=1}^{N} a_n y_n \kappa(\boldsymbol{x}, \boldsymbol{x}_n) + \sum_{n=1}^{N} a_n y_n, \tag{18.8}$$

where $< \cdot, \cdot >$ denotes the inner product in the RKHS, $\kappa(\cdot, \cdot)$ss the kernel associated with the RKHS, and we have used the kernel trick. The kernelized version of the perceptron algorithm is summarized in Algorithm 18.2.

[2] The readers who have not read Chapter 11, can bypass this section.

[3] Note that $\boldsymbol{\theta}$ may now be function, but we keep the same symbol as in the perceptron algorithm.

Algorithm 18.2 (The kernel perceptron algorithm).

- **For** $n = 1, 2, \ldots, N$, **Do**
  - $a_n = 0$
- **End For**
- **Repeat**
  - counter=0
  - **For** $i = 1, 2, \ldots, N$, **Do**
    - **If** $\left(y_n\left(\sum_{n=1}^{N} a_n y_n \kappa(\boldsymbol{x}_i, \boldsymbol{x}_n) + \sum_{n=1}^{N} a_n y_n\right) \leq 0\right)$ **Then**
      - $a_i = a_i + 1$
      - counter=counter+1
  - **End For**
- **Until** counter=0

18.3 FEED-FORWARD MULTILAYER NEURAL NETWORKS

A single neuron is associated with a hyperplane

$$H: \ \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_l x_l + \theta_0 = 0,$$

in the input (feature) space. Moreover, classification is performed via the nonlinearity, which fires an one or stays at zero, depending on which side of H a point lies. We will now show how to combine neurons, in a layer-wise fashion, to construct nonlinear classifiers. We will follow a simple constructive proof, which will unveil certain aspects of neural networks. These will be useful later on, when dealing with deep architectures.

As a starting point, we consider the case where the classes in the feature space are formed by unions of polyhedral regions. This is shown in Figure 18.3, for the case of the two-dimensional feature space. Polyhedral regions are formed as intersections of half-spaces, each one associated with a hyperplane. In Figure 18.3, there are three hyperplanes (straight lines in $\mathbb{R}^2$), indicated as H_1, H_2, H_3, giving rise to seven polyhedral regions. For each hyperplane, the $(+)$ and $(-)$ sides (half-spaces) are indicated. In the sequel, each one of the regions is labeled using a triplet of binary numbers, depending on which side it is located with respect to H_1, H_2, H_3. For example, the region labeled as (101), lies in the (+) side of H_1, the (-) side of H_2, and the (+) side of H_3. Figure 18.4a shows three neurons, realizing the three hyperplanes, H_1, H_2, H_3, of Figure 18.3, respectively. The associated outputs, denoted as y_1, y_2, y_3, form the label of the region in which the corresponding input pattern lies. Indeed, if the weights of the synapses have been appropriately set, then if a pattern originates from the region, say, (010), then the first neuron on the left will fire a zero ($y_1 = 0$), the second an one ($y_2 = 1$) and the right-most a zero ($y_3 = 0$). In other words, combining the outputs of the three neurons together, we have achieved a *mapping* of the input feature space into the three-dimensional space. More specifically, the mapping is performed on the vertices of the *unit* cube in $\mathbb{R}^3$, as shown in Figure 18.5. In the more general case, where p neurons are employed, the mapping will be on the vertices of the unit hypercube in $\mathbb{R}^p$. This layer of neurons comprises the first *hidden layer* of the network, which we are developing.

An alternative way to view this mapping is as a new representation of the input patterns in terms of code words. For three neurons/hyperplanes we can form 2^3 binary code-words, each corresponding to

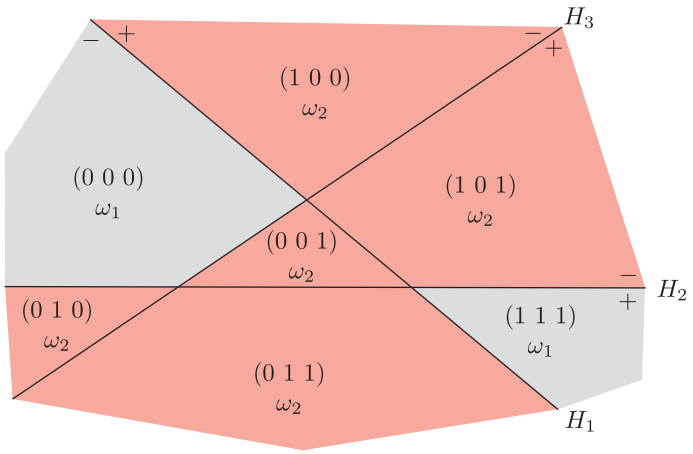

FIGURE 18.3

Classes are formed by the union of polyhedral regions. Regions are labeled according to the side on which they lie, with respect to the three lines, H_1, H_2, H_3. The number 1 indicates the (+) side and the 0 the (−) side. Class ω_1 consists of the union of the (000) and (111) regions.

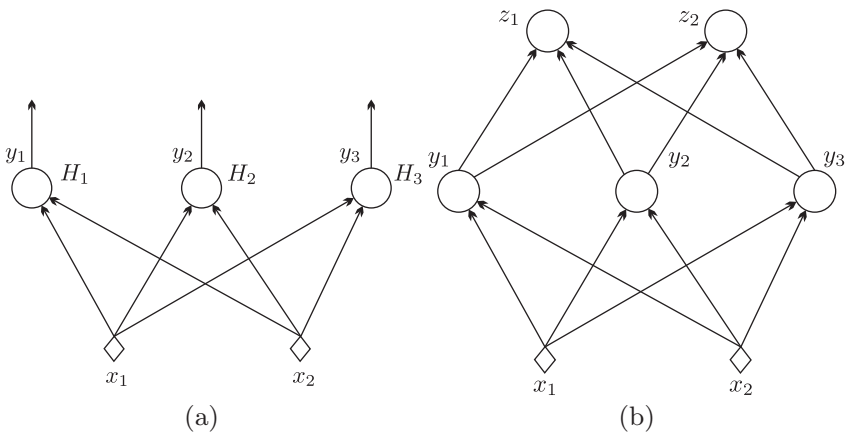

FIGURE 18.4

(a) The neurons of the first hidden layer are excited by the feature values applied at the input nodes and form the polyhedral regions. (b) The neurons of the second layer have as inputs the outputs of the first layer, and they thereby form the classes. To simplify the figure, the bias terms for each neuron are not shown.

a vertex of the unit cube, which can represent $2^3 - 1 = 7$ regions (there is one remaining vertex i.e., (110), which does not correspond to any region). Note, however, that this mapping encodes information concerning some *structure* of the input data; that is, information relating to how the input patterns are grouped together in the feature space in different regions.

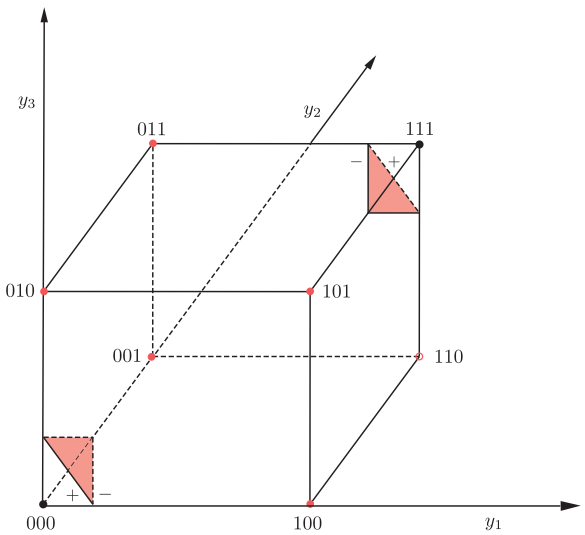

FIGURE 18.5

The neurons of the first hidden layer perform a mapping from the input feature space to the vertices of a unit hypercube. Each region is mapped to a vertex. Each vertex of the hypercube is now linearly separable from all the rest and can be separated by a hyperplane realized by a neuron. The vertex 110, denoted as an unshaded circle does not correspond to any region.

We will now use this new representation, as it is provided by the outputs of the neurons of the first hidden layer, as input which feeds the neurons of a second hidden layer, which is constructed as follows. We choose all regions that belong to one class. For the sake of our example in Figure 18.3, we select the two regions that correspond to class ω_1, that is, (000) and (111). Recall that all the points from these regions are mapped to the respective vertices of the unit cube in the $\mathbb{R}^3$. However, in this new transformed space, each one of the vertices is now *linearly separable* from the rest. This means that we can use a neuron/perceptron in the transformed space, which will place a single vertex in the (+) side and the rest in the (−) one, of the associated hyperplane. This is shown in Figure 18.5, where two planes are shown, which separate the respective vertices form the rest. Each of these planes is realized by a neuron, operating in $\mathbb{R}^3$, as shown in Figure 18.4b, where a second layer of *hidden* neurons has been added.

Note that the output z_1 of the left neuron will fire a 1 only if the input pattern originates from the region 000 and it will be at 0 for all other patterns. For the neuron on the right, the output z_2 will be 1 for all the patterns coming from region (111) and zero for all the rest. Note that this second layer of neurons has performed a second mapping, this time to the vertices of the unit rectangle in the $\mathbb{R}^2$. This mapping provides a new representation of the input patterns, and this representation encodes information related to the classes of the regions. Figure 18.6 shows the mapping to the vertices of the unit rectangle in the (z_1, z_2) space. Note that all the points originating from class ω_2 are mapped to (00) and the points from class ω_1 are mapped either to (10) or to (01). This is very interesting; by successive mappings, we have transformed our originally nonlinearly separable task, to one that is linearly separable. Indeed, the point (00) can be linearly separated from (01) and (10) and this can be

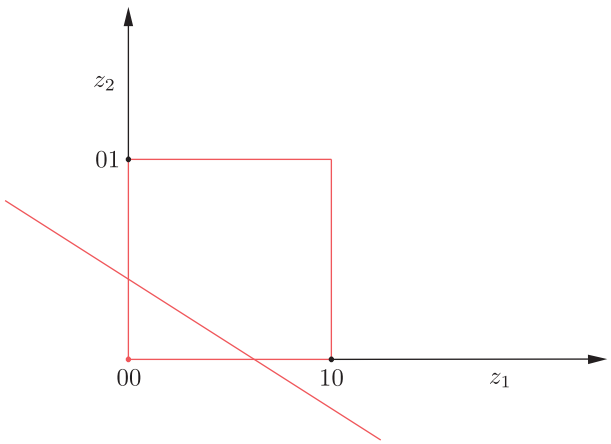

FIGURE 18.6

Patterns form class ω_1 are mapped either to (01) or to (10) and patterns from class ω_2 are mapped to (00). Thus the classes have now become linearly separable and can be separated via a straight line realized by a neuron.

realized by an extra neuron operating in the (z_1, z_2) space; it is known as the *output neuron*, because it provides the final classification decision. The final resulting network is shown is Figure 18.7. We call this network *feed-forward*, because information flows forward from the input to the output layer. It comprises the input layer, which is a nonprocessing one, two hidden layers (the term "hidden" is self-explained) and one output layer. We call such a NN an three-layer network, without counting the input layer of nonprocessing nodes.

We have constructively shown that a three-layer feed-forward NN can, in principle, solve *any* classification task whose classes are formed by the union of polyhedral regions. Although we focused on the two-class case, the generalization to multiclass cases is straightforward, by employing more output neurons depending on the number of classes. Note that in some cases, one hidden layer of nodes may be sufficient. This depends on whether the vertices on which the regions are mapped, are assigned to classes so linear separability is possible. For example, this would be the case if class ω_1 was the union of (000) and (100) regions. Then, these two corners could be separated from the rest via a single plane and a second hidden layer of neurons would not be required (check why). In any case, we will not take our discussion any further. The reason is that such a construction is important to demonstrate the power of building a multilayer NN, in analogy to what is happening in our brain. However, from a practical point of view, such a construction has not much to offer. In practice, when the data live in high-dimensional spaces, there is no chance of determining the parameters that define the neurons analytically to realize the hyperplanes, which form the polyhedral regions. Furthermore, in real life, classes are not necessarily formed by the union of polyhedral regions and more important classes do overlap. Hence, one needs to devise a training procedure based on a cost function.

All we will keep from our previous discussion is the structure of the multilayer network, and we will seek ways for estimating the unknown weights of the synapses and biases of the neurons. Moreover, from a conceptual point of view, we have to remember that each layer performs a mapping into a new

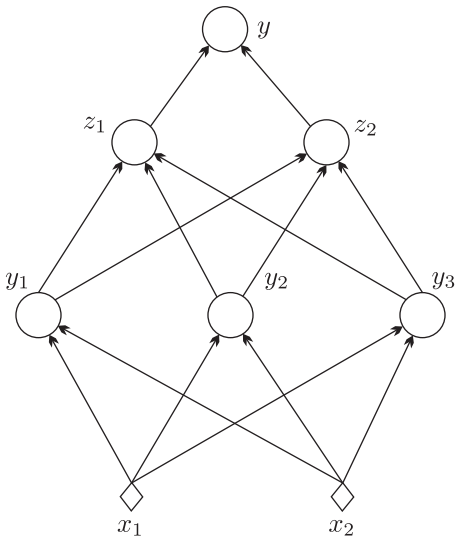

FIGURE 18.7

A three layer feed-forward neural network. It comprises the input (non-processing) layer, two hidden layers and one output layer of neurons. Such a three layer NN can solve *any* classification task, where classes are formed by unions of polyhedral regions.

space, and each mapping provides a different, hopefully more informative, representation of the input data, until the last layer where the task has been transformed into one that it is easy to solve.

18.4 THE BACKPROPAGATION ALGORITHM

A feed-forward neural network (NN) consists of a number of layers of neurons, and each neuron is determined by the corresponding set of synaptic weights and its bias term. From this point of view, an NN realizes a nonlinear parametric function, $\hat{y} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$ where $\boldsymbol{\theta}$ stands for all the weights/biases present in the network. Thus, training an NN seems not to be any different from training any other parametric prediction model. All that is needed is (a) a set of training samples, (b) a loss function, $\mathcal{L}(y, \hat{y})$, and (c) an iterative scheme, for example, gradient descent, to perform the optimization of the associated empirical loss,

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{N} \mathcal{L}\big(y_n, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\big).$$

The difficulty with training NNs lies in their multilayer structure that complicates the computation of the involved gradients, which are needed for the optimization. Moreover, the McCulloch-Pitts neuron involves the discontinuous Heaviside activation function, which is not differentiable. A first step in developing a practical algorithm for training an NN is to replace the Heaviside activation function with a differentiable approximation of it.

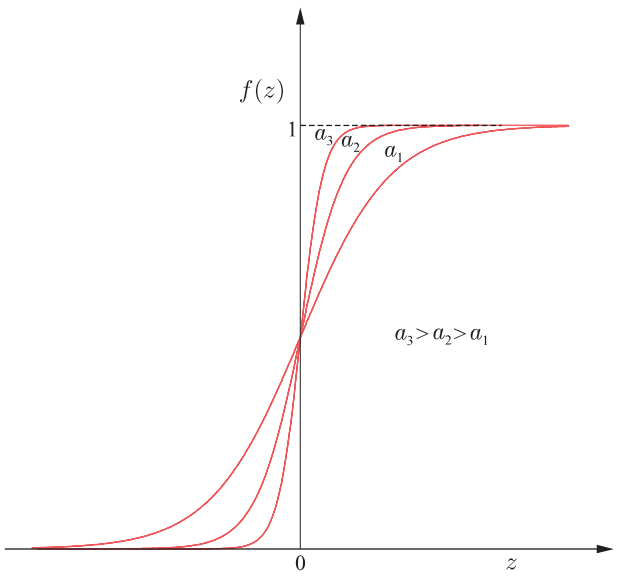

FIGURE 18.8

The logistic sigmoid function for different values of the parameter a.

The logistic sigmoid neuron: One possibility is to adopt the logistic sigmoid function, that is,

$$f(z) = \sigma(z) := \frac{1}{1 + \exp(-az)}. \tag{18.9}$$

The graph of the function is shown in Figure 18.8. Note that the larger the value of the parameter a, the closer the corresponding graph becomes to that of the Heaviside function. Another possibility would be to use

$$f(z) = a \tanh\left(\frac{cz}{2}\right), \tag{18.10}$$

where c and a are controlling parameters. The graph of this function is shown in Figure 18.9. Note that in contrast to the logistic sigmoid one, this is an antisymmetric function, that is, $f(-z) = -f(z)$. All these functions are also known as *squashing* functions, because they limit the output to a finite range of values.

18.4.1 THE GRADIENT DESCENT SCHEME

Having adopted a differentiable activation function, we are ready to proceed with developing the gradient descent iterative scheme for the minimization of the cost function. We will formulate the task in a general framework.

Let $(\boldsymbol{y}_n, \boldsymbol{x}_n)$, $n = 1, 2, \ldots, N$, be the set of training samples. Note that we have assumed multiple output variables, assembled as a vector. We assume that the network comprises L layers; $L - 1$ hidden and one output layers. Each layer consists of k_r, $r = 1, 2, \ldots, L$, neurons. Thus, the output vectors are

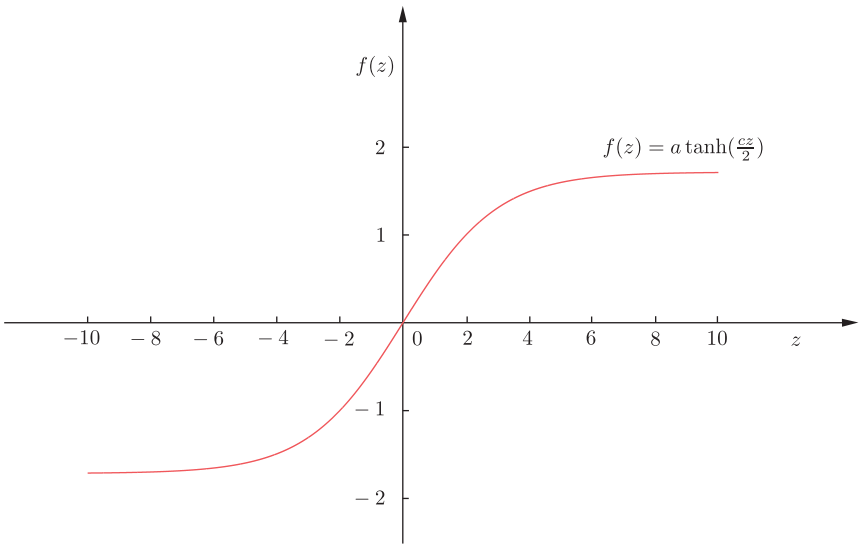

FIGURE 18.9

The hyperbolic tangent squashing function for $a = 1.7159$ and $c = 4/3$.

$$\boldsymbol{y}_n = [y_{n1}, y_{n2}, \ldots, y_{nk_L}]^T \in \mathbb{R}^{k_L}, \quad n = 1, 2, \ldots, N. \tag{18.11}$$

For the sake of the mathematical derivations, we also denote the number of input nodes as k_0; that is, $k_0 = l$, where l is the dimensionality of the input feature space.

Let $\boldsymbol{\theta}_j^r$ denote the synaptic weights associated with the jth neuron in the rth layer, with $j = 1, 2, \ldots, k_r$ and $r = 1, 2, \ldots, L$, where the bias term is included in $\boldsymbol{\theta}_j^r$, that is,

$$\boldsymbol{\theta}_j^r := [\theta_{j0}^r, \theta_{j1}^r, \ldots, \theta_{jk_{r-1}}^r]^T. \tag{18.12}$$

The synaptic weights link the respective neuron to all neurons in layer k_{r-1}, see Figure 18.10. The basic iterative step for the gradient descent scheme is written as

$$\boldsymbol{\theta}_j^r(new) = \boldsymbol{\theta}_j^r(old) + \Delta\boldsymbol{\theta}_j^r, \tag{18.13}$$

where

$$\Delta\boldsymbol{\theta}_j^r = -\mu \frac{\partial J}{\partial \boldsymbol{\theta}_j^r}\bigg|_{\boldsymbol{\theta}_j^r(old)}. \tag{18.14}$$

The parameter μ is the user-defined step-size (it can also be iteration-dependent) and J denotes the cost function. For example, if the squared error loss is adopted, we have

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{N} J_n(\boldsymbol{\theta}), \tag{18.15}$$

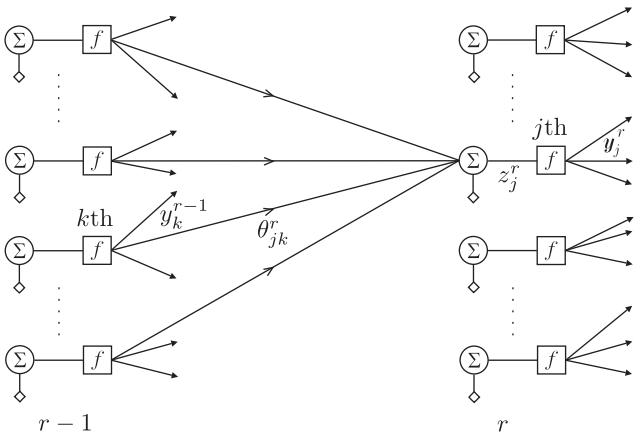

FIGURE 18.10

The links and the associated variables of the *j*th neuron at the *r*th layer.

and

$$J_n(\boldsymbol{\theta}) = \frac{1}{2}\sum_{k=1}^{k_L}\left(\hat{y}_{nk} - y_{nk}\right)^2, \tag{18.16}$$

where $\hat{y}_{nk},\ k = 1, 2, \ldots, k_L$, are the estimates provided at the corresponding output nodes of the network. We will consider them as the elements of a corresponding vector, $\hat{\boldsymbol{y}}_n$.

Computation of the gradients: Let z_{nj}^r denote the output of the linear combiner of the *j*th neuron in the *r*th layer at time instant n, when the pattern $\boldsymbol{x}_n$ is applied at the input nodes. Then, we can write that

$$z_{nj}^r = \sum_{m=1}^{k_{r-1}} \theta_{jm}^r y_{nm}^{r-1} + \theta_{j0}^r = \sum_{m=0}^{k_{r-1}} \theta_{jm}^r y_{nm}^{r-1} = \boldsymbol{\theta}_j^{rT} \boldsymbol{y}_n^{r-1}, \tag{18.17}$$

where by definition

$$\boldsymbol{y}_n^{r-1} := [1, y_{n1}^{r-1}, \ldots, y_{nk_{r-1}}^{r-1}]^T, \tag{18.18}$$

and $y_{n0}^r \equiv 1,\ \forall\ r, n$. For the neurons at the output layer, $r = L$, $y_{nm}^L = \hat{y}_{nm},\ m = 1, 2, \ldots, k_L$, and for $r = 1$, we have $y_{nm}^0 = x_{nm},\ m = 1, 2, \ldots, k_0$; that is, y_{nm}^0 are set equal to the input feature values.

Hence, we can now write that

$$\frac{\partial J_n}{\partial \boldsymbol{\theta}_j^r} = \frac{\partial J_n}{\partial z_{nj}^r}\frac{\partial z_{nj}^r}{\partial \boldsymbol{\theta}_j^r} = \frac{\partial J_n}{\partial z_{nj}^r}\boldsymbol{y}_n^{r-1}. \tag{18.19}$$

Let us now define

$$\delta_{nj}^r := \frac{\partial J_n}{\partial z_{nj}^r}. \tag{18.20}$$

Then we have

$$\boxed{\Delta\boldsymbol{\theta}_j^r = -\mu \sum_{n=1}^{N} \delta_{nj}^r \mathbf{y}_n^{r-1}, \quad r = 1, 2, \ldots, L.} \tag{18.21}$$

Computation of δ_{nj}^r: Here is where the heart of the backpropagation algorithm beats. For the computation of the gradients, δ_{nj}^r, one starts at the last layer, $r = L$, and proceeds *backwards* toward $r = 1$; this philosophy justifies the name given to the algorithm.

1. $r = L$: We have that

$$\delta_{nj}^L = \frac{\partial J_n}{\partial z_{nj}^L}. \tag{18.22}$$

For the squared error loss function,

$$J_n = \frac{1}{2} \sum_{k=1}^{k_L} \left(f(z_{nk}^L) - y_{nk} \right)^2. \tag{18.23}$$

Hence,

$$\begin{aligned} \delta_{nj}^L &= (\hat{y}_{nj} - y_{nj}) f'(z_{nj}^L), \\ &= e_{nj} f'(z_{nj}^L), \quad j = 1, 2, \ldots, k_L, \end{aligned} \tag{18.24}$$

where f' denotes the derivative of f, and e_{nj} is the error associated with the jth output variable at time n. Note that for the last layer, the computation of the gradient is straightforward.

2. $r < L$: Due to the successive dependence between the layers, the value of z_{nj}^{r-1} influences all the values z_{nk}^r, $k = 1, 2, \ldots, k_r$ of the next layer. Employing the chain rule for differentiation, we get

$$\delta_{nj}^{r-1} = \frac{\partial J_n}{\partial z_{nj}^{r-1}} = \sum_{k=1}^{k_r} \frac{\partial J_n}{\partial z_{nk}^r} \frac{\partial z_{nk}^r}{\partial z_{nj}^{r-1}}, \tag{18.25}$$

or

$$\frac{\partial J_n}{\partial z_{nj}^{r-1}} = \sum_{k=1}^{k_r} \delta_{nk}^r \frac{\partial z_{nk}^r}{\partial z_{nj}^{r-1}}. \tag{18.26}$$

However,

$$\frac{\partial z_{nk}^r}{\partial z_{nj}^{r-1}} = \frac{\partial \left(\sum_{m=0}^{k_{r-1}} \theta_{km}^r y_{nm}^{r-1} \right)}{\partial z_{nj}^{r-1}}, \tag{18.27}$$

where,

$$y_{nm}^{r-1} = f(z_{nm}^{r-1}), \tag{18.28}$$

which leads to,

$$\frac{\partial z_{nk}^r}{\partial z_{nj}^{r-1}} = \theta_{kj}^r f'(z_{nj}^{r-1}), \tag{18.29}$$

and combining with (18.25)–(18.26), we obtain the recursive rule

$$\boxed{\delta_{nj}^{r-1} = \left(\sum_{k=1}^{k_r} \delta_{nk}^r \theta_{kj}^r\right) f'(z_{nj}^{r-1}), \quad j = 1, 2, \ldots, k_{r-1}.} \tag{18.30}$$

For uniformity with (18.24), define

$$e_{nj}^{r-1} := \sum_{k=1}^{k_r} \delta_{nk}^r \theta_{kj}^r, \tag{18.31}$$

and we finally get

$$\delta_{nj}^{r-1} = e_{nj}^{r-1} f'(z_{nj}^{r-1}). \tag{18.32}$$

The only remaining computation is the derivative of f, which is easily shown to be equal to (Problem 18.2)

$$f'(z) = af(z)\big(1 - f(z)\big). \tag{18.33}$$

The derivation has been completed and the backpropagation scheme is summarized in Algorithm 18.3.

Algorithm 18.3 (The gradient descent backpropagation algorithm).

- Initialization
  - Initialize all synaptic weights and biases randomly with small, but not very small, values.
  - Select step size μ.
  - Set $y_{nj}^0 = x_{nj}$, $j = 1, 2, \ldots, k_0 = l$, $n = 1, 2, \ldots, N$.
- **Repeat**; Each repetition completes an epoch.
  - **For** $n = 1, 2, \ldots, N$, **Do**
    - **For** $r = 1, 2, \ldots, L$, **Do**; Forward computations.
      - **For** $j = 1, 2, \ldots, k_r$, **Do**
        Compute z_{nj}^r from (18.17).
        Compute $y_{nj}^r = f(z_{nj}^r)$.
      - **End For**
    - **End For**
    - **For** $j = 1, 2, \ldots, k_L$, **Do**
      - Compute δ_{nj}^L from (18.24).
    - **End For**
    - **For** $r = L, L-1, \ldots, 2$, **Do**; Backward computations.
      - **For** $j = 1, 2, \ldots, k_r$, **Do**
        Compute δ_{nj}^{r-1} from (18.32).
      - **End For**
    - **End For**
  - **End For**
  - **For** $r = 1, 2, \ldots, L$, **Do**; Update the weights.

- **For** $j = 1, 2, \ldots, k_r$, **Do**
  - Compute $\Delta\boldsymbol{\theta}_j^r$ from (18.21)
  - $\boldsymbol{\theta}_j^r = \boldsymbol{\theta}_j^r + \Delta\boldsymbol{\theta}_j^r$
- **End For**
- **End For**

- **Until** a stopping criterion is met.

The backpropagation algorithm can claim a number of fathers. The popularization of the algorithm is associated with the classical paper [75], where the derivation of the algorithm is provided. However, the algorithm had been derived much earlier in [97]. The idea of backpropagation also appears in [14] in the context of optimal control.

Remarks 18.2.

- A number of criteria have been suggested for terminating the backpropagation algorithm. One possibility is to track the value of the cost function, and stop the algorithm when this gets smaller than a preselected threshold. An alternative path is to check for the gradient values and stop when these become small; this means that the values of the weights do not change much from iteration to iteration, see, for example, [48].
- As it is the case with all gradient descent schemes, the choice of the step-size, μ, is very critical; it has to be small to guarantee convergence, but not too small; otherwise convergence speed slows down. The choice depends a lot on the specific problem at hand. Adaptive values of μ, whose value depends on the iteration, are more appropriate, and they will be discussed soon.
- Due to the highly nonlinear nature of the NN problem, the cost function in the parameter space is in general of a complicated form and there are many local minima, where the algorithm can be trapped. If such a local minimum is deep enough, the obtained solution can be acceptable. However, this may not be the case and the solution can be trapped in a shallow minimum resulting in a bad solution. In practice, one reinitializes randomly the weights a number of times and keeps the best solution. Initialization has to be performed with care; we discuss this later on. A more recent direction for initialization will be discussed in Section 18.8, in the context of deep learning.
- *Pattern-by-pattern operation*: The scheme discussed in Algorithm 18.3 is of the *batch* type of operation, where the weights are updated once per epoch; that is, after all N training patterns have been presented to the algorithm. The alternative route is the *pattern-by-pattern/online* mode of operation; for this case, the weights are updated at every time instant when a new pattern appears in the input. An intermediate way, where the update is performed every $N_1 < N$ samples, has also been considered, which is referred to as a *mini-batch* mode of operation. Batch and mini-batch modes of operation have an averaging effect on the computation of the gradients. In [78], it is advised to add a small white noise sequence to the training data, which may have a beneficial effect for the algorithm to escape from a poor local minimum.

  The pattern-by-pattern mode leads to a less smooth convergence trajectory; however, such a randomness may have the advantage of helping the algorithm to escape from a local minimum. To exploit randomness even further in the pattern-by-pattern mode, it is advisable that prior to the pass of a new epoch, the sequence in which data are presented to the algorithm is randomized, see, for example, [28]. This has no meaning in the batch mode, because updates take place once all data have been considered. In practice, the pattern-by-pattern version of the backpropagation seems to converge faster and give better solutions.

Online versions exploit better the training set, when redundancies in the data are present or training samples are very similar. Averaging, as it is done in the batch mode, wastes resources, because averaging the contribution to the gradient of similar patterns does not add much information. In contrast, in the pattern-by-pattern mode of operation, all examples are equally exploited, inasmuch as an update takes place for each one of the patterns.

Speeding up the convergence rate

The basic gradient descent scheme inherits all the advantages (low computational demands) and all the disadvantages (slow convergence rate) of the gradient descent algorithmic family, as it was first presented in this book in Chapter 5. To speed up the convergence rate, a large research effort has been invested in the late 1980s and early 1990s and a number of variants of the basic gradient descent backpropagation scheme have been proposed. In this section, we provide some directions that seem to be more popular in practice.

Gradient descent with a momentum term: One way to improve the convergence rate, while remaining within the gradient descent rationale, is to employ the so-called *momentum term* [24, 99]. The correction term in (18.21) is now modified as

$$\Delta\boldsymbol{\theta}_j^r(new) = a\Delta\boldsymbol{\theta}_j^r(old) - \mu\sum_{n=1}^{N}\delta_{nj}^r\boldsymbol{y}_n^{r-1}, \tag{18.34}$$

where a is the momentum factor. In other words, the algorithm takes into account the correction used in the previous iteration step as well as the current gradient computations. Its effect is to increase the step size, in regions where the cost function exhibits low curvature. Assuming that the gradient is approximately constant over, say I, successive iterations, it can be shown (Problem 18.3) that using the momentum term the updates are equivalent to

$$\Delta\boldsymbol{\theta}_j^r(I) \approx -\frac{\mu}{1-a}\boldsymbol{g}, \tag{18.35}$$

where $\boldsymbol{g}$ is the gradient value over the I successive iteration steps. Typical values of a are in the range of 0.1 to 0.8. It has been reported that the use of a momentum term can speed up the convergence rate up to a factor of two [79]. Experience seems to suggest that the use of a momentum factor helps the batch mode of operation more than the online version.

Iteration-dependent step-size: A heuristic variant of the previous backpropagation versions results if the step-size is left to vary as iterations progress. A rule is to change its value according to whether the cost function in the current iteration step is larger or smaller compared to the previous one. Let $J^{(i)}$ be the computed cost value at the current iteration. Then if $J^{(i)} < J^{(i-1)}$, the learning rate is increased by a factor of r_i. If, on the other hand, the new value is larger than the previous one by a factor larger than c, then the learning rate is reduced by a factor of r_d. Otherwise, the same value is kept. Typical values for the involved parameters are $r_i = 1.05$, $r_d = 0.7$, and $c = 1.04$. For iteration steps where the value of the cost increases, it is advisable to set the momentum factor equal to zero. Another possibility is not to perform the update whenever the cost increases. Such techniques, also known as *adaptive momentum*, are more appropriate for batch processing, because for online versions the values of the cost tend to oscillate from iteration to iteration.

Using different step-size for each weight: It is beneficial for improving the convergence rate, to employ a *different step-size* for each individual weight; this gives the freedom to the algorithm to exploit better the dependence of the cost function on each direction in the parameter space. In [46], it is suggested to increase the learning rate, associated with a weight, if the respective gradient value has the same sign for two successive iterations. Conversely, the learning rate is decreased if the sign changes, because this is indicative of possible oscillation.

Some practical hints

Training an NN still has a lot of practical engineering flavor compared to mathematical rigorousness. In this section, I will present some practical hints that experience has shown to be useful in improving the performance of the backpropagation algorithm; see, for example, [51] for a more detailed discussion.

Preprocessing the input features/variables: It is advisable to preprocess the input variables so they have (approximately) zero mean over the training set. Also, one should scale them so they all have similar variances, assuming that all variables are equally important. Their variance should also match the range of values of the activation (squashing) function. For example, for the hyperbolic tangent activation function, a variance of the order of one seems to be a good choice. Moreover, it is beneficial for the convergence of the algorithm if the input variables are uncorrelated. This can be achieved via an appropriate transform, for example, PCA.

Selecting symmetric activation functions: For the same reason that it is beneficial for the convergence when the inputs have zero mean, it is desirable that the outputs of the neurons assume equally likely positive and negative values. After all, the outputs of one layer become inputs to the next. To this end, the hyperbolic activation function in Eq. (18.10) can be used. Recommended values are $a = 1.7159$ and $c = 4/3$. These values guarantee that if the inputs are preprocessed as suggested before, that is, to be normalized to variances equal to one, then the variance at the output of the activation function is also equal to one and the respective mean value equal to zero.

Target values: The target values should be carefully chosen to be in line with the activation function used. The values should be selected to offset by some small amount the limiting value of the squashing function. Otherwise, the algorithm tends to push the weights to large values and this slows down the convergence; the activation function is driven to saturation, making the derivative of the activation function very small, which in turn renders small gradient values. For the hyperbolic tangent function, using the parameters discussed before, the choice of ± 1 for the target class labels seems to be the right one. Note that in this case, the saturation values are $a = \pm 1.7158$.

Initialization: The weights should be initialized randomly to values of small magnitude. If they are initialized to large values, then all activation functions will operate in their saturation point, making the gradients small, which slows down convergence. The effect on the gradients is the same, when the weights are initialized to very small values. Initialization must be done so that the operation in each neuron takes place in the (approximate) linear region of the graph of the activation function and not in the saturated one. It can be shown ([51], Problem 18.4) that if the input variables are preprocessed to zero mean and unit variance, and the tangent hyperbolic function is used with parameter values as discussed before, then the best choice for initializing the weights is to assign values drawn from a distribution with zero mean and standard deviation equal to

$$\sigma = m^{-1/2},$$

where m is the number of synaptic connections in the corresponding neuron.

18.4.2 BEYOND THE GRADIENT DESCENT RATIONALE

The other path to follow to improve upon the convergence rate of the gradient descent-based backpropagation algorithm, at the expense of increased complexity, is to resort to schemes that involve, in one way or another, information related to the second order derivatives. We have already discussed such families in this book, for example, the Newton family introduced in Chapter 6. For each one of the available families, a backpropagation version can be derived to serve the needs of the NN training. We will not delve into details, because the concept remains the same as that discussed for the gradient descent. The difference is that now second order derivatives have to be propagated backwards. The interested reader can look at the respective references and also in [12, 16, 26, 51, 102] for more details.

In [4, 47, 48] schemes based on the conjugate gradient philosophy have been developed and members of the Newton family have been proposed in, for example, [6, 69, 95]. In all these schemes, the computation of the elements of the Hessian matrix, that is,

$$\frac{\partial^2 J}{\partial \theta_{jk}^r \partial \theta_{j'k'}^{r'}},$$

is required. To this end, various simplifying assumptions are employed in the different papers (see, also, Problems 18.5 and 18.6).

A popular algorithm, which is loosely based on Newton's scheme, has been proposed in [20], and it is known as the *quickprop* algorithm. It is a heuristic method that treats the synaptic weights as if they were quasi-independent. It then approximates the error surface, as a function of each weight, via a quadratic polynomial. If this has its minimum at a sensible value, the latter is used as the new weight for the iterations; otherwise, a number of heuristics are mobilized. A common formulation for the resulting updating rule is given by,

$$\Delta\theta_{ij}^r(\text{new}) = \begin{cases} a_{ij}^r(\text{new})\Delta\theta_{ij}^r(\text{old}), & \text{if } \Delta\theta_{ij}^r(\text{old}) \neq 0, \\ -\mu \dfrac{\partial J}{\partial \theta_{ij}^r}, & \text{if } \Delta\theta_{ij}^r(\text{old}) = 0, \end{cases} \tag{18.36}$$

where

$$a_{ij}^r(\text{new}) = \min \left\{ \frac{\dfrac{\partial J(\text{new})}{\partial \theta_{ij}^r}}{\dfrac{\partial J(\text{old})}{\partial \theta_{ij}^r} - \dfrac{\partial J(\text{new})}{\partial \theta_{ij}^r}}, a_{\max}^r \right\}, \tag{18.37}$$

with typical values of the parameters used being $0.01 \leq \mu \leq 0.6$, and $a_{\max} \approx 1.75$. An algorithm similar in spirit with the quickprop has been proposed in [70].

In practice, when large networks and data sets are involved, simpler methods, such as carefully tuned gradient descent schemes, seem to work better than more complex second order techniques. The latter can offer improvements in smaller networks, especially in the context of regression tasks. The careful tuning of NNs, especially when they are large, is of *paramount importance*. The deep learning techniques to be discussed soon, when used as part of a pre-training phase of NNs, can be seen as an attempt for well-tuned initialization.

18.4.3 SELECTING A COST FUNCTION

As we have already commented, a feed-forward NN belongs to the more general class of parametric modeling; thus, in principle, any loss function we have met so far in this book can be employed to replace the least-squares one. Over the years, certain loss functions have gained in popularity in the context of NNs for classification tasks.

If one adopts as targets the $0, 1$ values, then the true and predicted values, $y_{nm}, \hat{y}_{nm}$, $n = 1, 2, \ldots, N$, $m = 1, 2, \ldots, k_L$, can be interpreted as probabilities and a commonly used cost function is the *cross-entropy*, which is defined as,

$$\boxed{J = -\sum_{n=1}^{N}\sum_{k=1}^{k_L}\left(y_{nk}\ln\hat{y}_{nk} + (1-y_{nk})\ln(1-\hat{y}_{nk})\right): \quad \text{Cross-Entropy Cost,}} \tag{18.38}$$

which takes its minimum value when $y_{nk} = \hat{y}_{nk}$, which for binary target values is equal to zero.

An interpretation of the cross-entropy cost comes from the following observation: the vector of target values, $\boldsymbol{y}_n \in \mathbb{R}^{k_L}$, has a single element equal to one, which indicates the class of the corresponding input pattern, $\boldsymbol{x}_n$; the rest of the elements are zero. Viewing each component, $\hat{y}_{nm}$, as the probability of obtaining a one at the respective node (class), then the probability $P(\boldsymbol{y}_n)$ is given by

$$P(\boldsymbol{y}_n) = \prod_{k=1}^{k_L}(\hat{y}_{nk})^{y_{nk}}(1-\hat{y}_{nk})^{1-y_{nk}}. \tag{18.39}$$

Then, it is straightforward to see that the cross-entropy cost function in (18.38) is the negative log-likelihood of the training samples. It can be shown (Problem 18.7) that, the cross-entropy function depends on the relative errors and not on the absolute errors, as is the case in the LS loss; thus, small and large error values are equally weighted during the optimization. Furthermore, it can be shown that the cross-entropy belongs to the so-called well-formed loss functions, in the sense that if there is a solution that classifies correctly all the training data, the gradient descent scheme will find it [2]. In [80], it is pointed out that the cross-entropy loss function may lead to improved generalization and faster training for classification, compared to the LS loss.

An alternative cost results if the similarity between y_{nk} and $\hat{y}_{nk}$ is measured in terms of the *relative entropy* or KL divergence,

$$\boxed{J = -\sum_{n=1}^{N}\sum_{k=1}^{k_L} y_{nk}\ln\frac{\hat{y}_{nk}}{y_{nk}}: \quad \text{Relative Entropy Cost.}} \tag{18.40}$$

Although we have interpreted the outputs of the nodes as probabilities, there is no guarantee that these add to one. This can be enforced by selecting the activation function in the last layer of nodes to be

$$\boxed{\hat{y}_{nk} = \frac{\exp(z_{nk}^L)}{\sum_{m=1}^{k_L}\exp(z_{nm}^L)}: \quad \text{Softmax Activation Function,}} \tag{18.41}$$

which is known as the *softmax activation* function [13]. It is easy to show that in this case, δ_{nj}^{L} required by the backpropagation algorithm is equal to $\hat{y}_j - y_j$ (Problem 18.9). A more detailed discussion on various cost functions can be found in, for example, [88].

18.5 PRUNING THE NETWORK

A crucial factor in training NNs is to decide the size of the network. The size is directly related to the number of weights to be estimated. We know that in any parametric modeling method, if the number of free parameters is large enough with respect to the number of training data, overfitting will occur.

Concerning feed-forward neural networks, two issues are involved. The first concerns the number of layers and the other the number of neurons per layer. As we will discuss in Section 18.8, a number of factors support the use of more than two hidden layers. However, experience has shown that trying to train such NNs via algorithms inspired by the backpropagation philosophy alone, will fail to obtain a reasonably good solution, due to the complicated shape of the cost function in the parameter space. Thus, in practice, one has to use at most two hidden layers. Otherwise, it seems that more sophisticated training techniques have to be adopted. Coming to the second issue, there is no theoretically supported model to assist the prediction of the number of neurons per layer. In practice, the most common technique is to start with a large enough number of neurons and then use a regularization technique to push the less informative weights to low values. A number of different regularization approaches have been proposed over the years. A brief presentation and some guidelines are given in the sequel.

Weight decay: This path refers to a typical cost function regularization via the Euclidean norm of the weights. Instead of minimizing a cost function, $J(\boldsymbol{\theta})$, its regularized version is used, such that,

$$J'(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda ||\boldsymbol{\theta}||^2. \tag{18.42}$$

Although this simple type of regularization helps in improving the generalization performance of the network, and it can be sufficient for some cases, in general it is not the most appropriate way to go. We have already discussed in Chapter 3 in the context of ridge regression that, involving the bias terms in the regularizing norm is not a good practice, because it affects the translation invariant property of the estimator. A more sensible way to regularize is to remove the bias terms from the norm. Moreover, it is even better if one groups the parameters of different layers together and employs different regularizing constants for each group.

Weight elimination: Instead of employing the norm of the weights, another approach involves more general functions for the regularization term, that is,

$$J'(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda h(\boldsymbol{\theta}). \tag{18.43}$$

For example, in [96] the following is used

$$h(\boldsymbol{\theta}) = \sum_{k=1}^{K} \frac{\theta_k^2}{\theta_h^2 + \theta_k^2}, \tag{18.44}$$

where K is the total number of the weights involved and θ_h is a preselected threshold value. A careful look at this function reveals that if $\theta_k < \theta_h$ the penalty term goes to zero very fast. In contrast, for values $\theta_k > \theta_h$, the penalty term tends to unity. In this way, less significant weights are pushed toward to zero. A number of variants of this method have also appeared, for example, [76].

Methods based on sensitivity analysis: In [50], the so-called *optimal brain damage* technique is proposed. A perturbation analysis of the cost function in terms of the weights is performed, via the second order Taylor expansion, that is,

$$\delta J = \sum_{i=1}^{K} g_i \delta\theta_i + \frac{1}{2}\sum_{i=1}^{K} h_{ii}\delta\theta_i^2 + \frac{1}{2}\sum_{i=1}^{K}\sum_{j=1, j\neq i}^{K} h_{ij}\delta\theta_i\delta\theta_j, \tag{18.45}$$

where,

$$g_i := \frac{\partial J}{\partial \theta_i} \quad h_{ij} := \frac{\partial^2 J}{\partial\theta_i\partial\theta_j}.$$

Then, assuming the Hessian matrix to be diagonal and if the algorithm operates near the optimum (zero gradient), we can approximately set

$$\delta J \approx \frac{1}{2}\sum_{i=1}^{K} h_{ii}\delta\theta_i^2. \tag{18.46}$$

The method works as follows:

- The network is trained using the backpropagation algorithm. After a few iteration steps, the training is frozen.
- The so called *saliencies*, defined as

  $$s_i = \frac{h_{ii}\theta_i^2}{2},$$

  are computed for each weight, and weights with a small saliency are removed. Basically, the saliency measures the effect on the cost function, if one removes (sets equal to zero) the respective weight.
- Training is continued and the process is repeated, until a stopping criterion is satisfied.

In [25], the full Hessian matrix is computed, giving rise to the *optimal brain surgeon* method.

Early stopping: An alternative primitive technique to avoid overfitting is the so-called *early stopping*. The idea is to stop the training when the test error starts increasing. Training the network over many epochs can lead the training error to converge to small values. However, this is an indication of overfitting rather than indicative of a good solution. According to the early stopping method, training is performed for some iterations and then it is frozen. The network, using the currently available estimates of the weights/biases, is used with a validation/test data set and the value of the cost function is computed. Then training is resumed, and after some iterations the previous process is repeated. When the value of the cost function, computed on the test set, starts increasing then training is stopped.

Remarks 18.3.

- *Weight Sharing*: One major issue encountered in many classification tasks is that of transformation invariance. This means that the classifier should classify correctly, independent of transformations performed on the input space, such as translation, rotation, and scaling. For example, the character 5 should "look the same" to an OCR system, irrespective of its position, orientation, and size. There are various ways to approach this problem. One is to choose appropriate feature vectors, which are invariant under such transformations, see, for example, [88]. Another way is to make the

classifier responsible for it in the form of *built-in constraints*. Weight sharing is such a constraint, which forces certain connections in the network to have the same weights, for example, [65].

- *Convolutional Networks*: This is a very successful example of networks that are built around the weight-sharing rationale. Convolutional networks have been inspired by the structural architecture of our visual system, for example, [42], and have been particularly successful in machine vision and optical character recognition schemes, where the inputs are images. Networks developed on these ideas are based on local connectivities between neurons and on hierarchically organized transformations of the image. Nodes form groups of two-dimensional arrays known as *feature maps*. Each node in a given map receives inputs from a specific window area of the previous layer, known as its *receptive field*. Translation invariance is imposed by forcing corresponding nodes in the same map, looking at different receptive fields, to share weights. Thus, if an object moves from one input receptive field to the other, the network responds in the same way.
  The first such architecture was proposed in [22] and it works in an unsupervised training mode. A supervised version of it was proposed in [49]; see also [52]. It turns out that such architectures closely resemble the physiology of our visual system, at least as far as the quick recognition of objects is concerned [77].
  A very interesting aspect of these networks is that they can have many hidden layers, without facing problems in their training. Training a general-purpose feed-forward NN with many layers, using standard backpropagation-type algorithms and random initialization, would be impossible; we will come back to this issue soon. Thus, convolutional networks are notable early successful examples of deep architectures.

18.6 UNIVERSAL APPROXIMATION PROPERTY OF FEED-FORWARD NEURAL NETWORKS

In Section 18.3, the classification power of a three-layer feed-forward NN, built around the McCulloch-Pitts neuron, was discussed. Then we moved on to employ smooth versions of the activation function, for the sake of differentiability. The issue now is whether we can say something more concerning the prediction power of such networks. It turns out that some strong theoretical results have been produced, which provide support for the use of NNs in practice, see, for example, [17, 23, 39, 45].

Let us consider a *two-layer* network, with one hidden layer and with a single output *linear* node. The output of the network is then written as

$$\hat{g}(\boldsymbol{x}) = \sum_{k=1}^{K} \theta_k^o f(\boldsymbol{\theta}_k^{hT} \boldsymbol{x}) + \theta_0^o, \tag{18.47}$$

where $\boldsymbol{\theta}_k^h$ denotes the synaptic weights and bias term defining the kth hidden neuron and the superscript "o" refers to the output neuron. Then, the following theorem holds true.

Theorem 18.1. *Let $g(\boldsymbol{x})$ be a continuous function defined in a compact*[4] *subset $S \subset \mathbb{R}^l$ and any $\epsilon > 0$. Then there is a two layer network with $K(\epsilon)$ hidden nodes of the form in Eq. (18.47), so that*

$$|g(\boldsymbol{x}) - \hat{g}(\boldsymbol{x})| < \epsilon, \quad \forall \boldsymbol{x} \in S. \tag{18.48}$$

[4] Closed and bounded.

In [5], it is shown that the approximation error decreases according to an $\mathcal{O}(1/K)$ rule. In other words, the input dimensionality does not enter into the scene and the error depends on the number of neurons used. The theorem states that a two-layer NN network is sufficient to approximate any continuous function; that is, it can be used to realize any nonlinear discriminant surface in a classification task or any nonlinear function for prediction in a general regression problem. This is a strong theorem indeed. However, what the theorem does not say is how big such a network can be in terms of the required number of neurons in the single layer. It may be that a very large number of neurons are needed to obtain a good enough approximation. This is where the use of more layers can be advantageous. Using more layers, the overall number of neurons needed to achieve certain approximation may be much smaller. We will come to this issue soon, when discussing deep architectures.

Remarks 18.4.

- *Extreme Learning Machines* (ELMs): These are single-layered feed-forward networks (SLFNs) with output of the form [40]:

$$g_K(\boldsymbol{x}) = \sum_{i=1}^{K} \theta_i^o f(\boldsymbol{\theta}_i^{hT}\boldsymbol{x} + b_i), \tag{18.49}$$

  where f is the respective activation function and K is the number of hidden nodes. The main difference with standard SLFNs is that the weights of each node (i.e., $\boldsymbol{\theta}_i^h$ and b_i) are generated *randomly*, whereas the weights of the output function (i.e., θ_i^o) are selected so that the squared error over the training points is minimized. This implies solving:

$$\min_{\boldsymbol{\theta}} \sum_{n=1}^{N} \left(y_n - g_K(\boldsymbol{x}_n)\right)^2, \tag{18.50}$$

  Hence, according to the ELM rationale, we do not need to compute the values of the parameters for the hidden layer. It turns out that such a training philosophy has a solid theoretical foundation, as convergence to a unique solution is guaranteed. It is interesting to note that, although the node parameters are randomly generated, for infinitely differentiable activation functions, the training error can become arbitrarily small, if K approaches N (it becomes zero if $K = N$). Furthermore, the universal approximation theorem ensures that for sufficiently large values of K and N, g_K can approximate any nonconstant piece-wise continuous function [41]. A number of variations and generalizations of this simple idea can be found in the respective literature. The interested reader is referred to, for example, [44, 67], for related reviews.

Example 18.1. In this example, the capability of a multilayer perceptron to classify nonlinearly separable classes is demonstrated. The classification task consists of two classes, each being the union of four regions in the two-dimensional space. Each region consists of normally distributed random vectors with statistically independent components and each with variance $\sigma^2 = 0.08$. The mean values are different for each of the regions. Specifically, the regions of the class denoted by a red $\circ$ (see Figure 18.11) are formed around the mean vectors

$$[0.4, 0.9]^T, [2.0, 1.8]^T, [2.3, 2.3]^T, [2.6, 1.8]^T$$

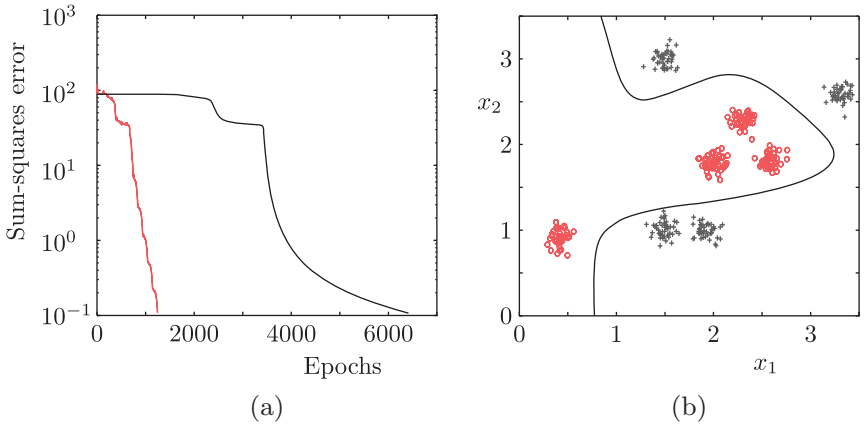

FIGURE 18.11

(a) Error convergence curves for the adaptive momentum (red line) and the momentum algorithms, for Example 18.1. Note that the adaptive momentum leads to faster convergence. (b) The classifier formed by the multilayer perceptron.

and those of the class denoted by a black + around the values

$$[1.5, 1.0]^T, [1.9, 1.0]^T, [1.5, 3.0]^T, [3.3, 2.6]^T.$$

A total of 400 training vectors were generated, 50 from each distribution. A multilayer perceptron with three neurons in the first and two neurons in the second hidden layer were used, with a single output neuron. The activation function was the logistic one with $a = 1$ and the desired outputs 1 and 0, respectively, for the two classes. Two different algorithms were used for the training, namely the momentum and the adaptive momentum; see discussion after Remarks 18.2. After some experimentation, the parameters employed were (a) for the momentum $\mu = 0.05$, $\alpha = 0.85$ and (b) for the adaptive momentum $\mu = 0.01$, $\alpha = 0.85$, $r_i = 1.05$, $c = 1.05$, $r_d = 0.7$. The weights were initialized by a uniform pseudorandom distribution between 0 and 1. Figure 18.11a shows the respective output error convergence curves for the two algorithms as a function of the number of epochs. The respective curves can be considered typical and the adaptive momentum algorithm leads to faster convergence. Both curves correspond to the batch mode of operation. Figure 18.11b shows the resulting classifier using the weights estimated from the adaptive momentum training.

A second experiment was conducted in order to demonstrate the effect of the pruning. Figure 18.12 shows the resulting decision lines separating the samples of the two classes, denoted by black and red ∘ respectively. Figure 18.12a corresponds to a multilayer perceptron with two hidden layers and 20 neurons in each of them, amounting to a total of 480 weights. Training was performed via the backpropagation algorithm. The overfitting nature of the resulting curve is readily observed. Figure 18.12b corresponds to the same multilayer perceptron trained with a pruning algorithm. Specifically, the method based on parameter sensitivity was used, testing the saliency values of the weights every 100 epochs and removing weights with saliency value below a chosen threshold. Finally, only 25 of the 480 weights survived and the curve is simplified to a straight line.

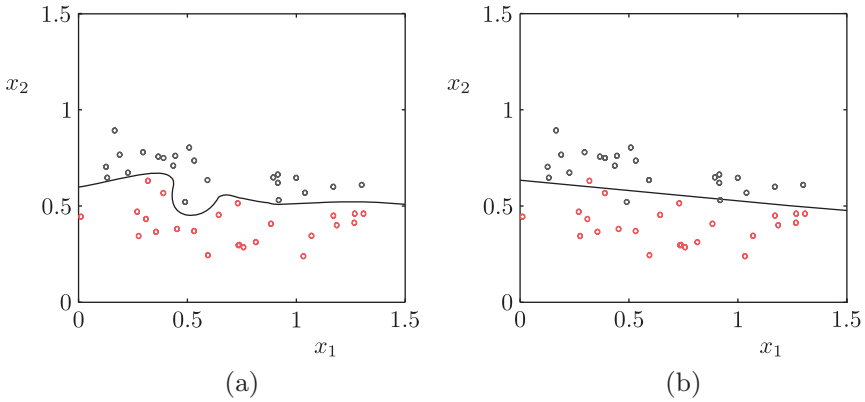

FIGURE 18.12

Decision curve (a) before pruning and (b) after pruning.

18.7 NEURAL NETWORKS: A BAYESIAN FLAVOR

In Chapter 12, the (generalized) linear regression and the classification tasks were treated in the framework of Bayesian learning. Because a feed-forward neural network realizes a parametric input-output mapping, $f_{\boldsymbol{\theta}}(\boldsymbol{x})$, there is nothing to prevent us from looking at the problem from a fully statistical point of view.

Let us focus on the regression task and assume that the noise variable is a zero mean Gaussian one. Then, the output variable, given the value of $f_{\boldsymbol{\theta}}(\boldsymbol{x})$, is described in terms of a Gaussian distribution,

$$p(y|\boldsymbol{\theta};\beta) = \mathcal{N}\left(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}),\beta^{-1}\right), \tag{18.51}$$

where β is the noise precision variable. Assuming successive training samples, $(y_n,\boldsymbol{x}_n)$, $n=1,2,\ldots,N$, to be independent, we can write

$$p(\boldsymbol{y}|\boldsymbol{\theta};\beta) = \prod_{n=1}^{N}\mathcal{N}\left(y_n|f_{\boldsymbol{\theta}}(\boldsymbol{x}_n),\beta^{-1}\right). \tag{18.52}$$

Adopting a Gaussian prior for $\boldsymbol{\theta}$, that is,

$$p(\boldsymbol{\theta};\alpha) = \mathcal{N}(\boldsymbol{\theta}|\mathbf{0},\alpha^{-1}I), \tag{18.53}$$

the posterior distribution, given the output values $\boldsymbol{y}$, can be written as

$$p(\boldsymbol{\theta}|\boldsymbol{y}) \propto p(\boldsymbol{\theta};\alpha)p(\boldsymbol{y}|\boldsymbol{\theta};\beta). \tag{18.54}$$

However, in contrast to Eq. (12.16), the posterior is not a Gaussian one, owing to the nonlinearity of the dependence on $\boldsymbol{\theta}$. Here is where complications arise and one has to employ a series of approximations to deal with it.

Laplacian approximation: The Laplacian approximation method, introduced in Chapter 12, is adopted to approximate $p(\boldsymbol{\theta}|\boldsymbol{y})$ to a Gaussian one. To this end, the maximum, $\boldsymbol{\theta}_{\text{MAP}}$, has to be computed,

which is carried out via an iterative optimization scheme. Once this is found, the posterior can be replaced by a Gaussian approximation, denoted as, $q(\boldsymbol{\theta}|\boldsymbol{y})$.

Taylor expansion of the neural network mapping: The final goal is to compute the predictive distribution,

$$p(y|\boldsymbol{x},\boldsymbol{y}) = \int p(y|f_{\boldsymbol{\theta}}(\boldsymbol{x}))q(\boldsymbol{\theta}|\boldsymbol{y})\,\mathrm{d}\boldsymbol{\theta}. \tag{18.55}$$

However, although the involved pdfs are Gaussians, the integration is intractable, because of the nonlinear nature of $f_{\boldsymbol{\theta}}$. In order to carry this out, a first order Taylor expansion is performed,

$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) \approx f_{\boldsymbol{\theta}_{\mathrm{MAP}}}(\boldsymbol{x}) + \boldsymbol{g}^T(\boldsymbol{\theta} - \boldsymbol{\theta}_{\mathrm{MAP}}), \tag{18.56}$$

where $\boldsymbol{g}$ is the respective gradient computed at $\boldsymbol{\theta}_{\mathrm{MAP}}$, which can be computed using backpropagation arguments. After this linearization, the involved pdfs become linear with respect to $\boldsymbol{\theta}$ and the integration leads to an approximate Gaussian predictive distribution as in Eq. (12.21).

For the classification, instead of the Gaussian pdf, the logistic regression model as in Section 13.7.1 of Chapter 13 is adopted and similar approximations as before are employed.

More on the Bayesian approach to NNs can be obtained in [56, 57]. In spite of their theoretical interest, the Bayesian approach has not been widely adopted in practice, compared to their backpropagation-based algorithmic relatives.

18.8 LEARNING DEEP NETWORKS

In our tour so far in this chapter, we have discussed various aspects of learning networks with more than two layers of nodes. The backpropagation algorithm, in its various formulations, was introduced as a popular scheme for training multilayer architectures. We also established some very important features of the multilayer perceptrons concerning their universal approximation property and also their power to solve any classification task comprising classes formed by the union of polyhedra regions in the input space. Two or three layers were, theoretically, enough to perform such tasks. Thus, it seems that everything has been said. Unfortunately (or maybe fortunately) this is far from the truth.

Multilayer perceptrons, after almost two decades of intense research, lost their initial glory and were superseded, to a large extent, by other techniques, such as kernel-based schemes, boosting and boosted trees, and Bayesian learning methods. A major reason for this loss of popularity was that their training can become difficult and often backpropagation-related algorithms are stuck in local minima. Although improvements can be obtained by trying different practical "tricks," such as multiple training using random initialization, still their generalization performance may not be competitive with other methods. This drawback becomes more severe if more than two hidden layers are used. The more layers one uses, the more difficult the training becomes and the probability to recover solutions corresponding to poor local minima is increased. As a matter of fact, efforts to use more than two hidden layers were soon abandoned.

In this section, we are going to focus on the following two issues:

- Is there any need for networks with more than two or three layers?
- Is there a training scheme, beyond or complementary to the backpropagation algorithm, to assist the optimization process to settle in a "good" local minimum, by extracting and exploiting more information from the input data?

Answers to both these points will be presented, starting with the first one.

18.8.1 THE NEED FOR DEEP ARCHITECTURES

In Section 18.3, we discussed how each layer of a neural network provides a different description of the input patterns. The input layer describes each pattern as a point in the feature space. The first hidden layer of nodes (using the Heaviside activation) forms a partition of the input space and places the input point in one of the regions, using a coding scheme of zeros and ones at the outputs of the respective neurons. This can be considered as a more abstract representation of our input patterns. The second hidden layer of nodes, based on the information provided by the previous layer, encodes information related to the classes; this is a further representation abstraction, which carries some type of "semantic meaning." For example, it provides the information of whether a tumor is malignant or benign, in a related medical application.

The previously reported hierarchical type of representation of the input patterns mimics the way that a mammal's brain follows to "understand" and "sense" the world around us; in the case of humans, this is the physical mechanism in the brain on which *intelligence* is built. The brain of the mammals is organized in a number of layers of neurons, and each layer provides a different representation of the input percept. In this way, different levels of abstraction are formed, via a hierarchy of transformations. For example, in the primate visual system, this hierarchy involves detection of edges, primitive shapes, and as we move to higher hierarchy levels, more complex visual shapes are formed, until finally a semantics concept is established; for example, a car moving in a video scene, a person sitting in an image. The cortex of our brain can be seen as a multilayer architecture with 5-10 layers dedicated only to our visual system, [77].

An issue that is now raised is whether one can obtain an equivalent input-output representation via a relatively simple functional formulation (such as the one implied by the support vector machines) or via networks with less than three layers of neurons/processing elements, maybe at the expense of more elements per layer.

The answer to the first point is yes, as long as the input-output dependence relation is simple enough. However, for more complex tasks, where more complex concepts have to be learned, for example, recognition of a scene in a video recording, language and speech recognition, the underlying functional dependence is of a very complex nature so that we are unable to express it analytically in a simple way.

The answer to the second point, concerning networks, lies in what is known as *compactness* of representation. We say that a network, realizing an input-output functional dependence, is compact if it consists of relatively few free parameters (few computational elements) to be learned/tuned during the training phase. Thus, for a given number of training points, we expect compact representations to result in better generalization performance.

It turns out that using networks with more layers, one can obtain more compact representations of the input-output relation. Although there are not theoretical findings for general learning tasks to prove such a claim, theoretical results from the theory of circuits of Boolean functions suggest that a function, which can compactly be realized by, say, k layers of logic elements, may need an exponentially large number of elements if it is realized via $k-1$ layers. Some of these results have been generalized and are valid for learning algorithms in some special cases. For example, the parity function with l inputs requires $\mathcal{O}(2^l)$ training samples and parameters to be represented by a Gaussian support vector machine, $\mathcal{O}(l^2)$ parameters for a neural network with one hidden layer, $\mathcal{O}(l)$ parameters and nodes for a multilayer network with $\mathcal{O}(\log_2 l)$ layers; see, for example, [7, 8, 64].

Such arguments may seem a bit confusing, because we have already stated that networks with two layers of nodes are universal approximators for a certain class of functions. However, this theorem

does not say how one can achieve this in practice. For example, any continuous function can be approximated arbitrarily close by a sum of monomials. Nevertheless, a huge number of monomials may be required, which is not practically feasible. In any learning task, we have to be concerned with what is feasibly "learnable" in a given representation. The interested reader may refer to, for example, [90] for a discussion on the benefits one is expected to get when using many-layer architectures.

Let us now elaborate a bit more on the aforementioned issues and also make bridges to schemes discussed in previous chapters. Recall from Chapter 11 that nonparametric techniques, modeling the input-output relation in RKH spaces, establish a functional dependence of the form,

$$f(\boldsymbol{x}) = \sum_{n=1}^{N} \theta_n \kappa(\boldsymbol{x}, \boldsymbol{x}_n) + \theta_0. \tag{18.57}$$

This can be seen as a network with one hidden layer, whose processing nodes perform kernel computations and the output node performs a linear combination. As already commented in Section 11.10.4, the kernel function, $\kappa(\boldsymbol{x}, \boldsymbol{x}_n)$, can be thought of as a measure of similarity between $\boldsymbol{x}$ and the respective training sample, $\boldsymbol{x}_n$. For kernels such as the Gaussian one, the action of the kernel function is of a local nature, in the sense that the contribution of $\kappa(\boldsymbol{x}, \boldsymbol{x}_n)$ in the summation tends to zero as the distance of $\boldsymbol{x}$ from $\boldsymbol{x}_n$ increases (the rate of decreasing influence depends on the variance σ^2 of the Gaussian). Thus, if the true input-output functional dependence undergoes fast variations, then a large number of such local kernels will be needed to model sufficiently well the input-output relation. This is natural, as one attempts to approximate a fast-changing function in terms of smooth bases of a local extent. Similar arguments hold true for the Gaussian processes discussed in Chapter 13. Besides the kernel methods, other widely used learning schemes are also of a local nature, as is the case for the decision trees, discussed in Chapter 7. This is because the input space is partitioned into regions via rules that are local for each one of the regions.

In contrast, assuming that these variations are not random in nature but that there exist underlying (unknown) regularities, resorting to models with a more compact representation, such as networks with many layers, one expects to learn the regularities and exploit them to improve the performance. As stated in [72], exploiting the regularities that are hidden in the training data is likely to design an excellent predictor for future events. The interested reader may explore more on these issues from the insightful tutorial [9].

From now on, we will refer to the number of layers in a network as the *depth* of the network. Networks with up to three (two hidden) layers, are known as *shallow*, whereas those with more than three are called *deep* networks. The main issue associated with a deep architecture is its training. As said before, the use of backpropagation fails to provide satisfactory generalization performance. A breakthrough that paved the way for training such large networks was proposed in [34].

18.8.2 TRAINING DEEP NETWORKS

A new philosophy for training deep networks was proposed in [34]. The main idea is to *pre-train* each layer, via an *unsupervised* learning algorithm, one layer at a time, in a *greedy-like* rationale. Different options are open for selecting the unsupervised learning scheme. The most popular is the one suggested in [34] that builds upon a special type of Boltzmann machines known as the *restricted*

Boltzmann machine (RBM), which will be treated in more detail in the next subsection. Needless to say that this is a new field of research with an intense happening in various application disciplines. New techniques are still being developed, and new experimental evidence is added to the existing one. Thus, the terrain may change as new information concerning these networks becomes available. Our goal is to point out the major ideas and techniques that are currently used. The reader must be flexible and engaged in following new developments in this fast-growing area.

Figure 18.13 presents a block diagram of a deep neural network with three hidden layers. The vector of the input random variables is denoted as $\mathbf{x}$ and those associated with the hidden ones as $\mathbf{h}^i$, $i = 1, 2, 3$. The vector of the output nodes is denoted as $\mathbf{y}$. Pre-training evolves in a sequential fashion, starting from the weights connecting the input nodes to the nodes of the first hidden layer. As we will see soon, this is achieved by maximizing the likelihood of the observed samples of the input observations, $\boldsymbol{x}$, and treating the variables associated with the first layer as hidden ones. Once the weights corresponding to the first layer have been computed, the respective nodes are allowed to fire an output value and a vector

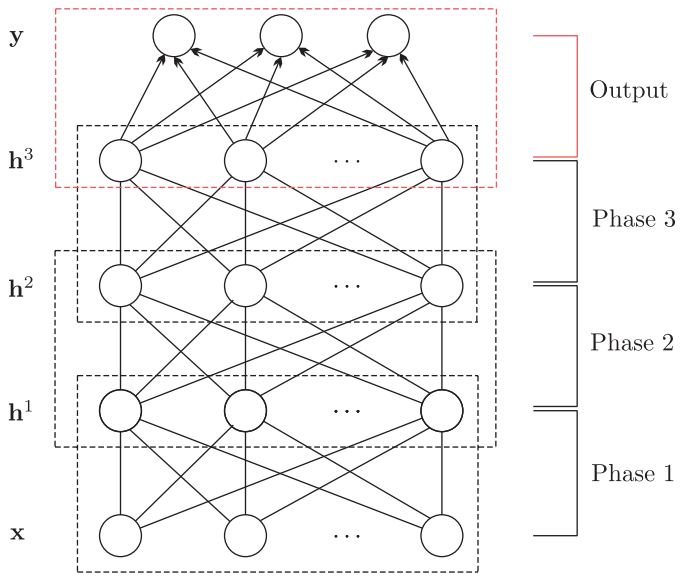

FIGURE 18.13

Block diagram of a deep neural network architecture, with three hidden layers and one output layer. The vector of the input random variables at the input layer is denoted as $\mathbf{x}$. The vector of the variables associated with the nodes of the ith hidden layer is denoted as $\mathbf{h}^i$, $i = 1, 2, 3$. The output variables are denoted as $\mathbf{y}$. At each phase of the pre-training, the weights associated with one hidden layer are computed, one at a time. For the network of the figure, comprising three hidden layers, pre-training consists of three stages of unsupervised learning. Once pre-training of the hidden units has been completed, the weights associated with the output nodes are pre-trained via a supervised learning algorithm. During the final fine-tuning, all the parameters are estimated via a supervised learning rule, such as the backpropagation scheme, using as initial values those obtained during pre-training.

of values, $\boldsymbol{h}^1$, is formed. This is the reason that a generative model for the unsupervised pre-training is adopted (such as the RBM), to be able to generate *in a probabilistic way* outputs at the hidden nodes. These values are in turn used as observations for the pre-training of the next hidden layer, and so on.

Once pre-training has been completed, a supervised learning rule, such as the backpropagation, is then employed to obtain the values of weights leading to the output nodes, as well as to fine tune the weights associated with the hidden layers, using as initial weight values those obtained during the pre-training phase.

Before proceeding into mathematical details, some further comments regarding the adopted approach can be helpful in better understanding the philosophy behind this type of training procedure.

We can interpret each one of the hidden layers as creating a feature vector and our deep architecture as a scheme for learning a *hierarchy of features*. The higher the layer is the higher the abstraction of the representation, associated with the respective feature vector. Using many layers of nodes, we leave the network to decide and generate a hierarchy of features, in an effort to capture the regularities underlying the data. Using deep architectures, one can provide as input to the network a "coding" scheme, which is as close as possible to the raw data, without being necessary for the designer to intervene and generate the features. This is very natural, because in complex tasks, which have to learn and predict nontrivial concepts, it is difficult for a human to know and generate good features that encode efficiently the relevant information, which resides in the data; grasping this information is vital for the generalization power of the model during prediction. Hence, the idea in deep learning is to leave the feature generation task, as much as possible, to the network itself.

It seems that unsupervised learning is a way to discover and unveil information hidden in the data, by learning the underlying regularities and the statistical structure of the data. In this way, pre-training can be thought of as a data-dependent *regularizer* that pushes the unknown parameters to regions where good solutions exist, by exploiting the extra information acquired by the unsupervised learning, see, for example, [19]. It is true to say that, some more formal and theoretically pleasing arguments, which can justify the good generalization performance obtained by such networks, are still to come.

Distributed representations

A notable characteristic of the features generated internally, layer by layer, in a multilayer neural network is that they offer what is known in machine learning as *distributed representation* of the input patterns. Some of the node outputs are 1 and the rest are 0. Interpreting each node as a feature, that provides information with respect to the input patterns, a distributed representation is spread among all these possible features, which are not mutually exclusive. In the antipodal of such a representation would be to have a single neuron firing each time. Moreover, it turns out that such a distributed representation is sparse, because only a few of the neurons are active each time. This is in line with what we believe happens in the human brain, where at each time less than 5% of the neurons, in each layer, fire, and the rest remain inactive.

Sparsity is another welcome characteristic, which is strongly supported by the more general learning theory, for example, [91]. Following information theoretic arguments, it can be shown that to get good generalization performance, the number of bits needed to encode the whole training set should be small with respect to the number of training data. Moreover, sparsity offers the luxury of encoding different examples with different binary codes, as required in many applications.

At the other extreme of representation is the one offered by local methods, where a different model is attached to each region in space and parameters are optimized locally. However, it turns out that

distributed representations can be exponentially more compact, compared to local representations. Take as an example the representation of integers in the interval $[1, 2, \ldots, N]$. One way is to use a vector of length N and for each integer to set the respective position equal to 1. However, a more efficient way in terms of the number of bits would be to employ a distributed representation; that is, use a vector of size $\log_2 N$ and encode each integer via ones and zeros positioned to express the number as a sum of powers of two. An early discussion on the benefits of distributed representation in learning tasks can be found in [30]. A more detailed treatment of these issues is provided in [9].

18.8.3 TRAINING RESTRICTED BOLTZMANN MACHINES

A *restricted Boltzmann machine* (RBM) is a special type of the more general class of Boltzmann machines (BMs), which were introduced in Chapter 15, [1, 82]. Figure 18.14 shows the probabilistic graphical model corresponding to an RBM. There are no connections among nodes of the same layer. Moreover, the upper level comprises nodes corresponding to hidden variables and the lower level consists of visible nodes. That is, observations are applied to the nodes of the lower layer only. Following the general definition of a Boltzmann machine, the joint distribution of the involved random variables is of the form,

$$P(v_1, \ldots, v_J, h_1, \ldots, h_I) = \frac{1}{Z} \exp\big(-E(\boldsymbol{v}, \boldsymbol{h})\big), \tag{18.58}$$

where we have used different symbols for the J visible (v_j, $j = 1, 2, \ldots J$) and the I hidden variables (h_i, $i = 1, 2, \ldots, I$). The *energy* is defined in terms of a set of unknown parameters,[5] that is,

$$P(v_1, \ldots, v_J, h_1, \ldots, h_I) = \frac{1}{Z} \exp\big(-E(\boldsymbol{v}, \boldsymbol{h})\big), \tag{18.59}$$

where b_i and c_j are the bias terms for the hidden and visible nodes, respectively. The normalizing constant is obtained as,

$$Z = \sum_{\boldsymbol{v}} \sum_{\boldsymbol{h}} \exp\big(-E(\boldsymbol{v}, \boldsymbol{h})\big). \tag{18.60}$$

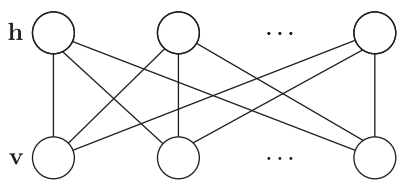

FIGURE 18.14

An RBM is an undirected graphical model with no connections among nodes of the same layer. In the context of the deep networks, the lower level comprises visible nodes and the upper layer consists of hidden nodes only.

[5] Compared to the notation used in Section 15.4.2 we use a negative sign. This is only to suit better the needs of the section, and it is obviously of no importance for the derivations.

We will focus on discrete variables, hence the involved distributions are probabilities. More specifically, we will focus on variables of a binary nature, that is, $\mathrm{v}_j,\ \mathrm{h}_i \in \{0,1\},\ j=1,\ldots,J,\ i=1,\ldots,I$. Observe from Eq. (18.59) that, in contrast to a general Boltzmann machine, only products between hidden and visible variables are present in the energy term.

The goal in this section is to derive a scheme for training an RBM; that is, to learn the set of unknown parameters, $\theta_{ij},\ b_i,\ c_j$, which will be collectively denoted as Θ, $\boldsymbol{b}$, and $\boldsymbol{c}$, respectively. The method to follow is to maximize the log-likelihood, using N observations of the visible variables, denoted as $\boldsymbol{v}_n,\ n=1,2,\ldots,N$, where

$$\boldsymbol{v}_n := [v_{1n},\ldots,v_{Jn}]^T,$$

is the vector of the corresponding observations at time n. We will say that the visible nodes are *clamped* on the respective observations. Once we establish a training scheme for an RBM, we will see how this mechanism can be embedded into a deep network for pre-training.

The corresponding (average) log-likelihood is given by,

$$\begin{aligned} L(\Theta,\boldsymbol{b},\boldsymbol{c}) &= \frac{1}{N}\sum_{n=1}^{N}\ln P(\boldsymbol{v}_n;\Theta,\boldsymbol{b},\boldsymbol{c}) \\ &= \frac{1}{N}\sum_{n=1}^{N}\ln\Big(\frac{1}{Z}\sum_{\boldsymbol{h}}\exp\big(-E(\boldsymbol{v}_n,\boldsymbol{h};\Theta,\boldsymbol{b},\boldsymbol{c})\big)\Big) \\ &= \frac{1}{N}\sum_{n=1}^{N}\ln\Big(\sum_{\boldsymbol{h}}\exp\big(-E(\boldsymbol{v}_n,\boldsymbol{h};\Theta,\boldsymbol{b},\boldsymbol{c})\big)\Big) \\ &\quad -\ln\sum_{\boldsymbol{v}}\sum_{\boldsymbol{h}}\exp\big(-E(\boldsymbol{v},\boldsymbol{h})\big), \end{aligned}$$

where the index n in the energy refers to the respective observations onto which the visible nodes have been clamped, and Θ has explicitly been brought into the notation.

Taking the derivative of $L(\Theta,\boldsymbol{b},\boldsymbol{c})$ with respect to θ_{ij} (similar to the case with respect to b_i and c_j), and applying standard properties of derivatives, it is not difficult to show (Problem 18.10), that

$$\boxed{\frac{\partial L(\Theta,\boldsymbol{b},\boldsymbol{c})}{\partial\theta_{ij}} = \frac{1}{N}\sum_{n=1}^{N}\left(\sum_{\boldsymbol{h}}P(\boldsymbol{h}|\boldsymbol{v}_n)h_i v_{jn}\right) - \sum_{\boldsymbol{v}}\sum_{\boldsymbol{h}}P(\boldsymbol{v},\boldsymbol{h})h_i v_j,} \tag{18.61}$$

where we have used that,

$$P(\boldsymbol{h}|\boldsymbol{v}) = \frac{P(\boldsymbol{v},\boldsymbol{h})}{\sum_{\boldsymbol{h}'}P(\boldsymbol{v},\boldsymbol{h}')}.$$

The gradient in (18.61) involves two terms. The first one can be computed once $P(\boldsymbol{h}|\boldsymbol{v})$ is available; we will derive it shortly. Basically, this term is the mean *firing rate* or *correlation* when the RBM is operating in its clamped phase; often we call it the *positive* phase, and the term is denoted as $<\mathrm{h}_i\mathrm{v}_j>^+$. The second term is the corresponding correlation when the RBM is working in its *free running* or *negative* phase and it is denoted as $<\mathrm{h}_i\mathrm{v}_j>^-$. Thus, a gradient ascent scheme for maximizing the log-likelihood will be of the form,

$$\theta_{ij}(\text{new}) = \theta_{ij}(\text{old}) + \mu \left(< \mathrm{h}_i \mathrm{v}_j >^{+} - < \mathrm{h}_i \mathrm{v}_j >^{-} \right).$$

Before going any further, let's take a minute to justify why we have named the two phases of operation as positive and negative, respectively. These terms appear in the seminal papers on Boltzmann machines by Hinton and Senjowski [29, 31]. The first one, corresponding to the clamped condition, can be thought of as a form of a *Hebbian* learning rule. Hebb was a neurobiologist and stated the first ever (to my knowledge) learning rule [27]: "If two neurons on either side of a synapse are activated simultaneously, the strength of this synapse is selectively increased." Note that this is exactly the effect of the positive phase correlation in the parameter's update recursion. On the contrary, the effect of the negative phase correlation term is the opposite. Thus, the latter term can be thought of as a *forgetting* or *unlearning* contribution; it can be considered as a control condition of a purely "internal" nature (note that it does not depend on the observations), compared to the "external" information received from the environment (observations).

Computation of the conditional probabilities

From the respective definitions, we get that

$$P(\boldsymbol{h}|\boldsymbol{v}) = \frac{1}{Z} \frac{\exp\left(-E(\boldsymbol{v},\boldsymbol{h})\right)}{P(\boldsymbol{v})} = \frac{\exp\left(-E(\boldsymbol{v},\boldsymbol{h})\right)}{\sum_{\boldsymbol{h}'} \exp\left(-E(\boldsymbol{v},\boldsymbol{h}')\right)}, \tag{18.62}$$

and plugging in the definition of the energy in Eq. (18.59), we obtain

$$\begin{aligned} P(\boldsymbol{h}|\boldsymbol{v}) &= \frac{\exp\left(\sum_{i=1}^{I}\sum_{j=1}^{J}\theta_{ij}h_i v_j + \sum_{i=1}^{I} b_i h_i\right)\exp\left(\sum_{j=1}^{J} c_j v_j\right)}{\sum_{\boldsymbol{h}'}\exp\left(\sum_{i=1}^{I}\sum_{j=1}^{J}\theta_{ij}h_i' v_j + \sum_{i=1}^{I} b_i h_i'\right)\exp\left(\sum_{j=1}^{J} c_j v_j\right)} \\ &= \prod_{i=1}^{I} \frac{\exp\left(\sum_{j=1}^{J}\theta_{ij}v_j + b_i\right)h_i}{\sum_{h_i'}\left[\exp\left(\sum_{j=1}^{J}\theta_{ij}v_j + b_i\right)h_i'\right]}. \end{aligned} \tag{18.63}$$

The factorization is a direct consequence of the RBM modeling, where no connections among hidden nodes are present (Problem 18.11). The previous formula readily suggests that,

$$P(h_i|\boldsymbol{v}) = \frac{\exp\left(\sum_{j=1}^{J}\theta_{ij}v_j + b_i\right)h_i}{\sum_{h_i'}\left[\exp\left(\sum_{j=1}^{J}\theta_{ij}v_j + b_i\right)h_i'\right]}, \tag{18.64}$$

which for the binary case becomes

$$P(h_i = 1|\boldsymbol{v}) = \frac{\exp\left(\sum_{j=1}^{J}\theta_{ij}v_j + b_i\right)}{1 + \exp\left(\sum_{j=1}^{J}\theta_{ij v_j} + b_i\right)}, \tag{18.65}$$

which can compactly be written as

$$\boxed{P(h_i = 1|\boldsymbol{v}) = \text{sigm}\left(\sum_{j=1}^{J}\theta_{ij}v_j + b_i\right),} \tag{18.66}$$

and recalling the definition of the logistic sigmoid function (18.9), we have that

$$\text{sigm}(z) = 1 - \sigma(z).$$

Due to the symmetry involved in the defining equations, it can be similarly shown that,

$$\boxed{P(v_j = 1|\boldsymbol{h}) = \text{sigm}\Big(\sum_{i=1}^{I} \theta_{ij} h_i + c_j\Big).} \tag{18.67}$$

Contrastive divergence

To train the RBM, one has to obtain the positive and negative phase correlations. However, the computation of the latter is intractable. A way to approach it is via Gibbs sampling techniques (see Chapter 14). The fact that we know analytically the conditionals, $P(h_i|\boldsymbol{v})$ and $P(v_j|\boldsymbol{h})$, allows us to apply Gibbs sampling by sequentially drawing samples, that is, $\boldsymbol{h}^{(1)} \sim P(\boldsymbol{h}|\boldsymbol{v}^{(1)})$, $\boldsymbol{v}^{(2)} \sim P(\boldsymbol{v}|\boldsymbol{h}^{(1)})$, $\boldsymbol{h}^{(2)} \sim P(\boldsymbol{h}|\boldsymbol{v}^{(2)})$, and so on. However, one has to wait long until the chain converges to a distribution representative of the true one. This is a reason that such networks had not been widely used in practical applications. In [33, 34], the method known as *contrastive divergence* (CD) was introduced. The maximum likelihood loss function was approximated as a difference of two Kullback-Leibler divergences. The end result, from an algorithmic point of view, can be conceived as a stochastic approximation attempt, where expectations are replaced by samples. There is, however, a notable difference: no samples are available for the hidden variables. According to the CD method, these samples are *generated* via Gibbs sampling, starting the chain from the observations available for the visible nodes. The most important feature is that, in practice, only a few iterations of the chain are sufficient.

Following this rationale, a first primitive version of this algorithmic scheme can be cast as:

- Step 1: Start the Gibbs sampler at $\boldsymbol{v}^{(1)} := \boldsymbol{v}_n$ and generate samples for the hidden variables, that is,

$$\boldsymbol{h}^{(1)} \sim P(\boldsymbol{h}|\boldsymbol{v}^{(1)}).$$

- Step 2: Use $\boldsymbol{h}^{(1)}$ to generate samples for the visible nodes,

$$\boldsymbol{v}^{(2)} \sim P(\boldsymbol{v}|\boldsymbol{h}^{(1)}).$$

  These are known as *fantasy data*.
- Step 3: Use $\boldsymbol{v}^{(2)}$ to generate the next set of hidden variables,

$$\boldsymbol{h}^{(2)} \sim P(\boldsymbol{h}|\boldsymbol{v}^{(2)}).$$

The scheme based on these steps is known as CD-1, because only one up-down-up Gibbs sweep is used. If k such steps are employed, the resulting scheme is referred to as CD-k. Once the samples have been generated, the parameter update can be written as

$$\theta_{ij}(n) = \theta_{ij}(n-1) + \mu \left(h_i^{(1)} v_{jn} - h_i^{(2)} v_j^{(2)}\right). \tag{18.68}$$

Note that in the first term in the parenthesis the clamped value of the jth visible node is used; in the second one, we use the sample that is obtained after running the Gibbs sampling on the model itself. It is common to represent the contrastive divergence update rule as

$$\Delta\theta_{ij} \propto \langle \text{h}_i \text{v}_j \rangle_{\text{data}} - \langle \text{h}_i \text{v}_j \rangle_{\text{recon}}, \tag{18.69}$$

where the first expectation (over the hidden unit activations) is with respect to the data distribution and the second expectation is with respect to the distribution of the "reconstructed" data, via Gibbs sampling.

A more refined scheme results if the estimates of the gradients are not obtained via a single observation sample, but they are instead averaged over a number of observations. In this vein, the training input examples are divided in a number of disjoint chunks, each one comprising, say, L examples. These blocks of data are also known as *mini-batches*. The previous steps are performed for each observation, but now the update is carried out only once per block of L samples, by averaging out the obtained estimates of the gradient, that is,

$$\theta_{ij}^{(t)} = \theta_{ij}^{(t-1)} + \frac{\mu}{L}\sum_{l=1}^{L} g_{ij}^{(l)}, \quad i = 1, \ldots, I,\ j = 1, \ldots, J, \tag{18.70}$$

where

$$g_{ij}^{(l)} := h_i^{(1)} v_{j(l)} - h_i^{(2)} v_j^{(2)},$$

denotes the gradient approximation associated with the corresponding observation $v_{j(l)}$, $(l) \in \{1, 2, \ldots, N\}$, which is currently considered by the algorithm (and gives birth to the associated Gibbs samples). Recursion (18.70) can be written in a more compact form as

$$\Theta^{(t)} = \Theta^{(t-1)} + \frac{\mu}{L}\sum_{l=1}^{L} G_{ij}^{(l)}, \tag{18.71}$$

where

$$G_{ij}^{(l)} := \boldsymbol{h}^{(1)} \boldsymbol{v}_{(l)}^T - \boldsymbol{h}^{(2)} \boldsymbol{v}^{(2)T}.$$

Once all blocks have been considered, this corresponds to one epoch of training. The process continues for a number of successive epochs until a convergence criterion is met.

Another version of the scheme results if we replace the obtained samples of the hidden variables with their respective mean values. This, in turn, leads to estimates with lower variance [85]. This is in accordance to what is known as Rao-Blackwellization, where the generated samples are replaced by their expected values. In our current context, where the variables are of a binary nature, it is readily seen that

$$\mathbb{E}[\mathrm{h}_i^{(1)}] = P\big(h_i = 1 | v_{j(l)}\big) = \mathrm{sigm}\Big(\sum_{j=1}^{J} \theta_{ij}^{(t-1)} v_{j(l)} + b_i^{(t-1)}\Big), \tag{18.72}$$

$$\mathbb{E}[\mathrm{h}_i^{(2)}] = P\big(h_i = 1 | v_j^{(2)}\big) = \mathrm{sigm}\Big(\sum_{j=1}^{J} \theta_{ij}^{(t-1)} v_j^{(2)} + b_i^{(t-1)}\Big). \tag{18.73}$$

In this case, the updates become

$$\Theta^{(t)} = \Theta^{(t-1)} + \frac{\mu}{L}\sum_{l=1}^{L} G_{ij}^{(l)}, \tag{18.74}$$

$$G_{ij}^{(l)} := \mathbb{E}[\mathbf{h}^{(1)}] \boldsymbol{v}_{(l)}^T - \mathbb{E}[\mathbf{h}^{(2)}] \boldsymbol{v}^{(2)T}. \tag{18.75}$$

The updates of the bias terms are derived in a similar way (one can also assume that there are fictitious extra nodes of a fixed value +1, and incorporate the bias terms in θ_{ij}), and we get

$$\boldsymbol{b}^{(t)} = \boldsymbol{b}^{(t-1)} + \frac{\mu}{L}\sum_{l=1}^{L} \boldsymbol{g}_b^{(l)}, \tag{18.76}$$

$$\boldsymbol{g}_b^{(l)} := \mathbb{E}[\mathbf{h}^{(1)}] - \mathbb{E}[\mathbf{h}^{(2)}], \tag{18.77}$$

and

$$\boldsymbol{c}^{(t)} = \boldsymbol{c}^{(t-1)} + \frac{\mu}{L}\sum_{l=1}^{L} \boldsymbol{g}_c^{(l)}, \tag{18.78}$$

$$\boldsymbol{g}_c^{(l)} := \boldsymbol{v}_{(l)} - \boldsymbol{v}^{(2)}. \tag{18.79}$$

The resulting scheme, using the expected values version, is summarized in Algorithm 18.4.

Algorithm 18.4 (RBM learning via CD-1 for binary variables).

- Initialization
  - Initialize $\Theta^{(0)}, \boldsymbol{b}^{(0)}, \boldsymbol{c}^{(0)}$, randomly.
- **For** each epoch **DO**
  - **For** each block of size L **Do**
    - $G = O, \boldsymbol{g}_b = \mathbf{0}, \boldsymbol{g}_c = \mathbf{0}$; set gradients to zero.
    - **For** each $\boldsymbol{v}_n$ in the block **Do**
      - $\boldsymbol{h}^{(1)} \sim P(\boldsymbol{h}|\boldsymbol{v}_n)$
      - $\boldsymbol{v}^{(2)} \sim P(\boldsymbol{v}|\boldsymbol{h}^{(1)})$
      - $\boldsymbol{h}^{(2)} \sim P(\boldsymbol{h}|\boldsymbol{v}^{(2)})$
      - $G = G + \mathbb{E}[\mathbf{h}^{(1)}]\boldsymbol{v}_n^T - \mathbb{E}[\mathbf{h}^{(2)}]\boldsymbol{v}^{(2)}$
      - $\boldsymbol{g}_b = \boldsymbol{g}_b + \mathbb{E}[\mathbf{h}^{(1)}] - \mathbb{E}[\mathbf{h}^{(2)}]$
      - $\boldsymbol{g}_c = \boldsymbol{g}_c + \boldsymbol{v}_n - \boldsymbol{v}^{(2)}$
    - **End for**
    - $\Theta = \Theta + \frac{\mu}{L}G$
    - $\boldsymbol{b} = \boldsymbol{b} + \frac{\mu}{L}\boldsymbol{g}_b$
    - $\boldsymbol{c} = \boldsymbol{c} + \frac{\mu}{L}\boldsymbol{g}_c$
  - **End for**
  - If a convergence criterion is met, Stop.
- **End For**

Remarks 18.5.

- *Persistent contrastive divergence*: To present the contrastive divergence technique, we made a comment related to the stochastic approximation method; the main point is that it is a result of approximating the likelihood via the difference of two KL divergences. However, there is indeed a strong relation of the method with stochastic approximation arguments. As a matter of fact, a version very similar to contrastive divergence was derived in [101], in the context of the general Boltzmann machines, based entirely on stochastic approximation arguments for minimizing the log-likelihood cost function. The main idea is traced back to [62]. The difference with the

contrastive divergence lies in the fact that instead of resetting the chain to the data after each parameter update, the previous state of the chain is used for the next iteration of the algorithm. This initialization is often fairly close to the model distribution, even though the model has changed a bit in the parameter update. The algorithm is known as *persistent contrastive divergence* (PCD) to emphasize that the Markov chain is not reset between parameter updates. It can be shown that this algorithm generates a consistent estimator, even with one Gibbs cycle per iteration.

The PCD algorithm can be used to obtain gradient estimates in an *online* mode of operation or using mini-batches, using only a few training data points for the positive correlation term of each gradient estimate and only a few samples for the negative correlation term. It was demonstrated in [89] that this scheme can lead to enhanced performance, compared to CD.

A treatment of stochastic approximation techniques for minimizing the log-likelihood in the context of RBMs is given in [85]. This bridge paves the way of using the various "tricks" developed for the more general stochastic approximation methods, such as weight decaying or using of momentum terms to smooth out the convergence trajectory in the parameter space. Moreover, general tools from the stochastic approximation theory, concerning the convergence of such algorithms, can be mobilized.

- Since the advent of the contrastive divergence method, a number of papers have been dedicated to its theoretical analysis. In [15], it was shown that, in general, the fixed points of CD will differ from those of maximum likelihood; however, assuming the data is generated via an RBM, then asymptotically they both share the maximum likelihood solution as a fixed point. Conditions in [100] are derived to guarantee convergence of CD; however, they are difficult to satisfy in practice. An analysis of CD in terms of an expansion of the log-probability is given in [10]. In [43], contrastive divergence is related to the gradient of the log pseudo-likelihood of the model. In [84], the focus of the analysis is on the CD-1 and it is pointed out that this is not related to the gradient of any cost function. Furthermore, it is shown that a regularized CD update has a fixed point for a large class of regularization functions.

18.8.4 TRAINING DEEP FEED-FORWARD NETWORKS

Figure 18.13 illustrates a multilayer perceptron with three hidden layers. As is always the case with any supervised learning task, the kick-off point is a set of training examples, $(y_n, \boldsymbol{x}_n)$, $n = 1, 2, \ldots, N$. Training a deep multilayer perceptron, employing what we have said before, involves two major phases: (a) pre-training and (b) supervised fine-tuning. Pre-training the weights associated with hidden nodes involves unsupervised learning via the RBM rationale. Assuming K hidden layers, $\mathbf{h}^k$, $k = 1, 2, \ldots, K$, we look at them in pairs, that is, $(\mathbf{h}^{k-1}, \boldsymbol{h}^k)$, $k = 1, 2, \ldots, K$, with $\mathbf{h}^0 := \mathbf{x}$, being the input layer. Each pair will be treated as an RBM, in a hierarchical manner, with the outputs of the previous one becoming the inputs to the next. It can be shown, for example, in [34], that adding a new layer each time increases a variational lower bound on the log-probability of the training data.

Pre-training of the weights leading to the output nodes is performed via a supervised learning algorithm. The last hidden layer together with the output layer are not treated as an RBM, but as a one layer feed-forward network. In other words, the input to this *supervised* learning task are the features formed in the last hidden layer.

Finally, fine-tuning involves retraining in a typical backpropagation algorithm rationale, using the values obtained during pre-training for initialization. This is very important for getting a better

feeling and understanding of how deep learning works. The label information is used in the hidden layers *only* at the fine-tuning stage. During pre-training, the feature values in each layer grasp information related to the input distribution and the underlying regularities. The label information does not participate in the process of discovering the features. Most of this part is left to the unsupervised phase, during pre-training. Note that this type of learning can also work even if some of the data are unlabeled. Unlabeled information is useful, because it provides valuable extra information concerning the input data. As a matter of fact, this is at the heart of semisupervised learning, see, e.g., [88].

The methodology is summarized in Algorithm 18.5.

Algorithm 18.5 (Training deep neural networks).

- Initialization.
  - Initialize randomly all the weights for the hidden nodes, Θ^k, $\boldsymbol{b}^k$, $\boldsymbol{c}^k$, $k = 1, 2, \ldots, K$.
  - Initialize randomly the weights leading to the output nodes.
  - Set $\boldsymbol{h}^0(n) := \boldsymbol{x}_n,\ n = 1, 2, \ldots, N$.

Phase I: Unsupervised Pre-training of Hidden Units

- **For** $k = 1, 2, \ldots, K$, **Do**;
  - Treat $\boldsymbol{h}^{k-1}$ as visible nodes and $\boldsymbol{h}^k$ as hidden nodes to an RBM.
  - Train the RBM with respect to Θ^k, $\boldsymbol{b}^k$, $\boldsymbol{c}^k$, via Algorithm 18.4.
  - Use the obtained values of the parameters to generate in the layer, $\boldsymbol{h}^k$, N vectors, corresponding to the N observations.
    - Option 1:
      - $\boldsymbol{h}^k(n) \sim P(\boldsymbol{h}|\boldsymbol{h}^{k-1}(n)),\ n = 1, 2, \ldots, N$; Sample from the distribution.
    - Option 2:
      - $\boldsymbol{h}^k(n) = [P(h_1^k|\boldsymbol{h}^{k-1}(n), \ldots, P(h_{I_k}^k|\boldsymbol{h}^{k-1}(n)]^T,\ n = 1, 2, \ldots, N$; that is, propagate the respective probabilities. I_k is the number of nodes in the layer.
- **End For**

Phase II: Supervised Pre-training of Output Nodes

- Train the parameters of the pair $(\boldsymbol{h}^K, \boldsymbol{y})$, associated with the output layer, via any *supervised* learning algorithm. Treat $(y_n, \boldsymbol{h}^K(n)), n = 1, 2, \ldots, N$, as the training data.

Phase III: Fine-Tuning of All Nodes via Supervised Training

- Use the obtained values for all the parameters as initial values and train the whole network via the backpropagation, using $(y_n, \boldsymbol{x}_n),\ n = 1, 2, \ldots, N$ as training examples.

Remarks 18.6.

- Training a deep network has a lot of engineering flavor and one needs to acquire some experience by playing with such networks. This was also the case with the "shallow" networks treated in the beginning of the chapter, where some practical hints concerning the training of such networks were summarized in Section 18.4.1. Some of these hints can also be used for deep networks, when using the backpropagation algorithm, during the final fine-tuning phase. For training deep architectures, we also have to deal with some practical "tricks" concerning the unsupervised pre-training. A list of very useful suggestions are summarized in [36].

18.9 DEEP BELIEF NETWORKS

In line with the emphasis given in this chapter so far, we focused our discussion on deep learning on multilayer perceptrons for supervised learning. Our focus was on the information flow in the feed-forward or bottom-up direction. However, this is only part of the whole story. The other part concerns training *generative* models. The goal of such learning tasks is to "teach" the model to generate data. This is basically equivalent with learning probabilistic models that relate a set of variables, which can be observed, with another set of hidden ones. RBMs are just an instance of such models. Moreover, it has to emphasized that, RBMs can represent any discrete distribution if enough hidden units are used, [21, 55].

In our discussion up to now in this section, we viewed a deep network as a mechanism forming layer-by-layer features of features, that is, more and more abstract representations of the input data. The issue now becomes whether one can start from the last layer, corresponding to the most abstract representation, and follow a *top-down* path with the new goal of generating data. Besides the need in some practical applications, there is an additional reason to look at this reverse direction of information flow.

Some studies suggest that such top-down connections exist in our visual system to generate lower level features of images starting from higher level representations. Such a mechanism can explain the creation of vivid imagery during dreaming, as well as the disambiguating effect on the interpretation of local image regions by providing contextual prior information from previous frames, for example, [53, 54, 60].

A popular way to represent statistical generative models is via the use of probabilistic graphical models, which were treated in Chapters 15 and 16. A typical example of a generative model is that of sigmoidal networks, introduced in Section 15.3.4, which belong to the family of parametric Bayesian (belief) networks. A sigmoidal network is illustrated in Figure 18.15a, which depicts a directed acyclic

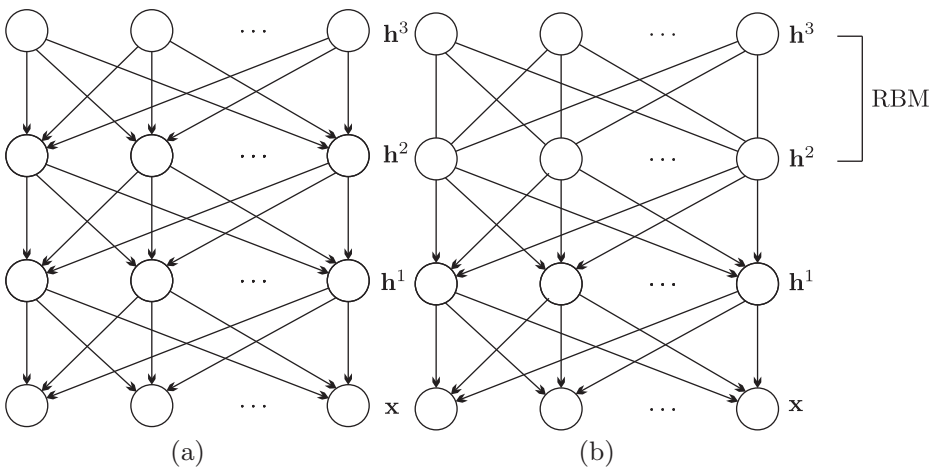

FIGURE 18.15

(a) A graphical model corresponding to a sigmoidal belief (Bayesian) network. (b) A graphical model corresponding to a deep belief network. It is a mixture of directed and undirected edges connecting nodes. The top layer involves undirected connections and it corresponds to an RBM.

graph (Bayesian). Following the theory developed in Chapter 15, the joint probability of the observed ($\boldsymbol{x}$) and hidden variables, distributed in K layers, is given by,

$$P(\boldsymbol{x}, \boldsymbol{h}^1, \ldots, \boldsymbol{h}^K) = P(\boldsymbol{x}|\boldsymbol{h}^1) \left(\prod_{k=1}^{K-1} P\left(\boldsymbol{h}^k|\boldsymbol{h}^{k+1}\right) \right) P(\boldsymbol{h}^K),$$

where the conditionals for each one of the I_k nodes of the kth layer are defined as,

$$P(h_i^k|\boldsymbol{h}^{k+1}) = \sigma\left(\sum_{j=1}^{I_{k+1}} \theta_{ij}^{k+1} h_j^{k+1} \right), \quad k = 1, 2, \ldots, K-1, \; i = 1, 2, \ldots, I_k.$$

A variant of the sigmoidal network was proposed in [34], which has become known as *deep belief network*. The difference with a sigmoidal one is that the top two layers comprise an RBM. Thus, it is a mixed type of network consisting of both directed as well as undirected edges. The corresponding graphical model is shown in Figure 18.15b. The respective joint probability of all the involved variables is given by

$$P(\boldsymbol{x}, \boldsymbol{h}^1, \ldots, \boldsymbol{h}^K) = P(\boldsymbol{x}|\boldsymbol{h}^1) \left(\prod_{k=1}^{K-2} P\left(\boldsymbol{h}^k|\boldsymbol{h}^{k+1}\right) \right) P\left(\boldsymbol{h}^{K-1}, \boldsymbol{h}^K\right). \tag{18.80}$$

It is known that learning Bayesian networks of relatively large size is intractable, because of the presence of converging edges (explaining away), see Section 15.3.3. To this end, one has to resort to variational approximation methods to bypass this obstacle, see Section 16.3. However, variational methods often lead to poor performance owing to simplified assumptions.

In [34], it is proposed that we employ the scheme summarized in Algorithm 18.5, Phase 1. In other words, all hidden layers, starting from the input one, are treated as RBMs, and a greedy layer-by-layer pre-training bottom-up philosophy is adopted. We should emphasize that the conditionals, which are recovered by such a scheme can only be thought of as approximations of the true ones. After all, the original graph is a directed one and is not undirected, as the RBM assumption imposes. The only exception lies at the top level, where the RBM assumption is a valid one.

Once the bottom-up pass has been completed, the estimated values of the unknown parameters are used for initializing another fine-tuning training algorithm, in place of the Phase III step of the Algorithm 18.5; however, this time the fine-tuning algorithm is an unsupervised one, as no labels are available. Such a scheme has been developed in [32] for training sigmoidal networks and is known as *wake-sleep* algorithm. The scheme has a variational approximation flavor, and if initialized randomly takes a long time to converge. However, using the values obtained from the pre-training for initialization, the process can significantly be speeded up [37]. The objective behind the wake-sleep scheme is to adjust the weights during the top-down pass, so as to maximize the probability of the network to generate the observed data.

Once training of the weights has been completed, data generation is achieved by the scheme summarized in Algorithm 18.6.

Algorithm 18.6 (Generating samples via a DBN).

- Obtain samples $\boldsymbol{h}^{K-1}$, for the nodes at level $K-1$. This can be done via running a Gibbs chain, by alternating samples, $\boldsymbol{h}^K \sim P(\boldsymbol{h}|\boldsymbol{h}^{K-1})$ and $\boldsymbol{h}^{K-1} \sim P(\boldsymbol{h}|\boldsymbol{h}^K)$. This can be carried out as explained in subsection 18.8.3, as the top two layers comprise an RBM. The convergence of the Gibbs chain can be speeded up by initializing the chain with a feature vector formed at the $K-1$ layer by one

of the input patterns; this can be done by following a bottom-up pass to generate features in the hidden layers, as the one used during pre-training.

- **For** $k = K-2, \ldots, 1$, **Do**; Top-down pass.
  - **For** $i = 1, 2, \ldots, I_k$, **Do**
    - $h_i^{k-1} \sim P\left(h_i | \boldsymbol{h}^k\right)$; Sample for each one of the nodes.
  - **End For**
- **End For**
- $\boldsymbol{x} = \boldsymbol{h}^0$; Generated pattern.

18.10 VARIATIONS ON THE DEEP LEARNING THEME

Besides the basic schemes, which have been presented so far, a number of variants have also been proposed. It is anticipated that in the years to come more and more versions will add on to the existing palette of methods. We will now report the main directions that were currently available at the time this book was published.

18.10.1 GAUSSIAN UNITS

In real-world applications such as speech recognition, data often consist of real-valued features, so the choice of binary visible units can be a modeling restriction. To deal with such types of data, we can use Gaussian visible units instead, that is, linear, real-valued units, with Gaussian noise [21], [58], [87]. If $v_j,\ j = 1, \ldots, J$ and $h_i,\ i = 1, \ldots, I$ are the (Gaussian) visible and (binary) hidden units of the RBM, respectively, the energy function, $E(\mathbf{v}, \mathbf{h})$, of the RBM becomes

$$E(\boldsymbol{v}, \boldsymbol{h}) = \sum_{j=1}^{J} \frac{(v_j - c_j)^2}{2\sigma_j^2} + \sum_{i=1}^{I} b_i h_i - \sum_{i,j} \theta_{ij} \frac{v_j}{\sigma_j} h_i, \tag{18.81}$$

where $c_j,\ j = 1, \ldots, J$ and $b_i,\ i = 1, \ldots, I$ are the biases of the visible and hidden units, respectively, $\sigma_j, j = 1, \ldots, J$, are the standard deviations of the Gaussian visible units, and $\theta_{ij},\ i = 1, \ldots, I, j = 1, \ldots, J$, are the weights connecting the visible and hidden units. The conditional probability, $P(h_i = 1 \mid \mathbf{v})$, of turning "on" a hidden unit is again the output of the logistic function, as in a standard RBM, that is,

$$P(h_i = 1 \mid \boldsymbol{v}) = \text{sigm}\left(b_i + \sum_{j=1}^{J} \theta_{ij} v_j\right), \quad i = 1, 2, \ldots, I. \tag{18.82}$$

However, the conditional pdf, $p(v_j \mid \boldsymbol{h})$, of a visible unit now becomes,

$$p(v_j \mid \boldsymbol{h}) = \mathcal{N}\left(c_j + \sum_{i=1}^{I} \theta_{ij} h_i, \sigma_j\right), \quad j = 1, 2, \ldots, J. \tag{18.83}$$

Ideally, the contrastive divergence algorithm should be modified, so as to be able to learn the σ_i's in addition to the c_i's, b_i's, and θ_{ij}'s (for a treatment of this topic, the reader is referred to [58]). However, in practice, the estimation of the σ_i's with the contrastive divergence algorithm is quite an unstable

procedure, and it is therefore preferable to normalize the data to zero mean and unit variance, prior to the RBM training stage. If this normalization step takes place, we no longer need to learn the variances, and we need to make only slight modifications to the contrastive divergence in Algorithm 18.4. More specifically, the sampling step

$$\boldsymbol{v}^{(2)} \sim P(\boldsymbol{v} \mid \boldsymbol{h}^{(1)})$$

is now performed by simply adding Gaussian noise of zero mean and unit variance, $\mathcal{N}(0, 1)$, to the accumulated input of each visible node. Furthermore, the output of each visible unit is no longer binary; however, this does not affect the sampling stages, $\boldsymbol{h}^{(1)}$ and $\boldsymbol{h}^{(2)}$ of the contrastive divergence algorithm due to the existence of the sigmoid function at the hidden units.

In practice, if Gaussian visible units are used, the step-size for weights and biases should be kept rather small during the training stage, compared to a standard RBM. For example, step-sizes of the order of 0.001 are not unusual. This is because we want to reduce the risk of the training algorithm to diverge, due to the linear nature of the visible units, which can receive large input values and do not possess a squashing function, which is capable of bounding the output to a predefined range of values.

It is also interesting to note that it is possible to have binary visible and Gaussian hidden units. This can be particularly useful for problems where one does not want to restrict the activation output of the hidden units to fall in the range $[0, 1]$. For example, this is the case in a deep encoder, used for dimensionality reduction purposes [35].

Finally, it is also possible to have the more general case of Gaussian units for both the visible as well as the hidden layers, although, in practice, such networks are very hard to train [58]. Note that in this more general case Eq. (18.81) becomes

$$E(\boldsymbol{v}, \boldsymbol{h}) = \sum_{j=1}^{J} \frac{(v_j - c_j)^2}{2\sigma_j^2} + \sum_{i=1}^{I} \frac{(h_i - b_i)^2}{2\sigma_i^2} - \sum_{i,j} \theta_{ij} \frac{v_j}{\sigma_j} \frac{h_i}{\sigma_i}. \tag{18.84}$$

18.10.2 STACKED AUTOENCODERS

Instead of building a deep network architecture by hierarchically training layers of RBMs, one can replace RBMs with autoencoders. *Autoencoders* have been proposed in [3, 75] as methods for dimensionality reduction. An autoencoder consists of two parts, the *encoder* and the *decoder*. The output of the encoder is the reduced representation of the input pattern, and it is defined in terms of a vector function,

$$\boldsymbol{f} : \boldsymbol{x} \in \mathbb{R}^l \longmapsto \boldsymbol{h} \in \mathbb{R}^m, \tag{18.85}$$

where,

$$h_i := f_i(\boldsymbol{x}) = \phi_e(\boldsymbol{\theta}_i^T \boldsymbol{x} + b_i^e), \quad i = 1, 2, \ldots, m, \tag{18.86}$$

with ϕ_e being the activation function; the latter is usually taken to be the logistic sigmoid function, $\phi_e(\cdot) = \sigma(\cdot)$.

The decoder is another function $\boldsymbol{g}$,

$$\boldsymbol{g} : \boldsymbol{h} \in \mathbb{R}^m \longmapsto \hat{\boldsymbol{x}} \in \mathbb{R}^l, \tag{18.87}$$

where

$$\hat{x}_j = g_j(\boldsymbol{h}) = \phi_d(\boldsymbol{\theta}_j^{'T}\boldsymbol{h} + b_j^d),\ j = 1, 2, \ldots, l. \tag{18.88}$$

The activation ϕ_d is, usually, taken to be either the identity (linear reconstruction) or the logistic sigmoid one. The task of training is to estimate the parameters,

$$\Theta := [\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_m,],\ \boldsymbol{b}^e,\ \Theta^{'} := [\boldsymbol{\theta}_1^{'}, \ldots, \boldsymbol{\theta}_l^{'}],\ \boldsymbol{b}^d.$$

It is common to assume that $\Theta^{'} = \Theta^T$. The parameters are estimated so the reconstruction error, $\boldsymbol{e} = \boldsymbol{x} - \hat{\boldsymbol{x}}$, over the available input samples are to be minimum in some sense. Usually, the least-squares cost is employed, but other choices are also possible. Regularized versions, involving a norm of the parameters, is also a possibility, for example, [71]. If the activation ϕ_e is chosen to be the identity (linear representation), and $m < l$ (to avoid triviality), the autoencoder is equivalent with the PCA technique [3]. PCA is treated in more detail in Chapter 19.

Another version of autoencoders results if one adds noise to the input [92, 93]. This is a stochastic counterpart, known as the *denoising autoencoder*. For reconstruction, the uncorrupted input is employed. The idea behind this version is that by trying to undo the effect of noise, one captures statistical dependencies between inputs. More specifically, in [92], the corruption process randomly sets some of the inputs (as many as half of them) to zero. Hence, the denoising autoencoder is forced to predict the missing values from the nonmissing ones, for randomly selected subsets of missing patterns.

Training a deep multilayer perceptron employing autoencoders consists of the following phases:

- Phase 1: Train the first hidden layer of nodes as an autoencoder; that is, by minimizing an adopted reconstruction error.
- Following the same rationale as for the RBMs, the hidden units' outputs of the autoencoder are used as inputs to feed the layer above. Training is done by treating the two layers as an autoencoder.
- Keep adding as many layers as is required by the depth of the network.
- The output of the last hidden layer is then used as input to the top output layer. The associated parameters are estimated in a supervised manner, using the available labels. Note that this is the first time during pre-training that one uses the label information.
- Employ an algorithm, for example, backpropagation for fine-tuning.

In [35], a different technique for fine tuning is suggested. The hierarchy of autoencoders is unfolded (i.e., both encoder and decoder are used) to reproduce a reconstruction of the input and an algorithm is used to minimize the error.

18.10.3 THE CONDITIONAL RBM

The *conditional restricted Boltzmann machine* (CRBM) [87] is an extension to the standard RBM, capable of modeling temporal dependencies among successive feature vectors (assuming that the training set consists of a set of feature sequences). Figure 18.16 presents the structure of a CRBM, where the hidden layer consists of binary stochastic neurons as is also the case with the standard RBM. However, it can be seen that there exist multiple layers of visible nodes, which, in the case of the figure, correspond to frames $\boldsymbol{v}^{t-2}$, $\boldsymbol{v}^{t-1}$, $\boldsymbol{v}^t$. Each visible layer corresponds to the feature vector at the respective time instant and consists of linear (Gaussian) units with zero mean and unity variance.

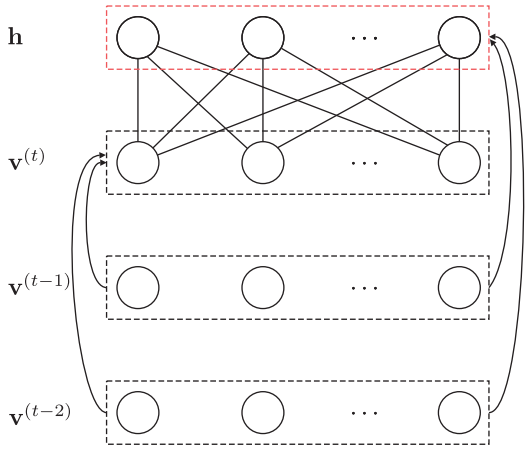

FIGURE 18.16

Structure of a conditional restricted Boltzmann machine.

The visible layers, which correspond to past time instants, are linked with *directional* links to the hidden layer and to the visible layer representing the tth (current) frame. Note that the tth layer of visible nodes is linked to the hidden layer with *undirected* connections, as in a standard RBM. The links (autoregressive weights) among visible nodes model the short-term temporal structure of the sequence of feature vectors, whereas the hidden units model longer (mid-term) characteristics of the feature sequence. As a result, the visible layers of previous time instants introduce a dynamically changing bias to the nodes in $\boldsymbol{v}^t$ and $\boldsymbol{h}$.

To proceed, let θ_{ij} be the undirectional weight connecting v_j^t with h_i, α_{ki}^{t-q} the directional weight connecting v_k^{t-q} with v_i^t, and d_{ij}^{t-q} the directional weight connecting v_j^{t-q} with h_i, where $q = 1, 2, \ldots, Q$ and Q is the length of the temporal context. The probability of turning the hidden unit h_i on is computed as follows [87]:

$$P(h_i \mid \boldsymbol{v}^t, \boldsymbol{v}^{t-1}, \ldots, \boldsymbol{v}^{t-Q}) = \text{sigm}\left(b_i + \sum_j \theta_{ij} v_j^t + \sum_{q=1}^{Q} \sum_j d_{ij}^{t-q} v_j^{t-q}\right), \tag{18.89}$$

where b_i is the bias of the ith hidden unit, and the last summation term is the dynamically changing bias of the ith hidden node, due to the temporal context.

The conditional pdf, of the linear unit v_j^t, is given by

$$p(v_j^t \mid \boldsymbol{h}, \boldsymbol{v}^t, \boldsymbol{v}^{t-1}, \ldots, \boldsymbol{v}^{t-Q}) = \mathcal{N}\Big(c_j + \sum_i \theta_{ij} h_j + \sum_{q=1}^{Q} \sum_k v_k^{t-q} \alpha_{kj}^{t-q}, 1\Big) \tag{18.90}$$

where c_j is the bias of the jth visible unit and $\mathcal{N}(\mu, 1)$ is the normal distribution with mean μ and unit variance. The last summation term plays the role of the dynamically changing bias of the visible nodes.

Despite the insertion of the aforementioned two types of directed connections, it is still possible to use the contrastive divergence algorithm to update both the undirected (θ_{ij}) and directed connections (α_{ki}^{t-q} and d_{ij}^{t-q}) [87]. The learning rule for θ_{ij} is the same as in the case of an RBM with binary hidden units. Specifically, recalling the notation used in Eq. (18.69), we can write

$$\Delta\theta_{ij} \propto \langle \mathrm{h}_i \mathrm{v}_j \rangle_{data} - \langle \mathrm{h}_i \mathrm{v}_j \rangle_{recon}, \tag{18.91}$$

where the angular brackets denote expectations with respect to the distributions of the data and reconstructed data, respectively. In this line of thinking, the learning rules for the hidden biases (b_j) and visible biases (c_i), are

$$\Delta b_i \propto \langle \mathrm{h}_i \rangle_{data} - \langle \mathrm{h}_i \rangle_{recon} \tag{18.92}$$

and

$$\Delta c_j \propto v_j - \langle \mathrm{v}_j \rangle_{\mathrm{recon}}, \tag{18.93}$$

respectively. The learning rule for the directed connections, d_{ij}^{t-q}, is

$$\Delta d_{ij}^{t-q} \propto v_j^{t-q} \left(\langle \mathrm{h}_i \rangle_{\mathrm{data}} - \langle \mathrm{h}_i \rangle_{\mathrm{recon}} \right), \tag{18.94}$$

Finally, the learning rule for the autoregressive weights, α_{ki}^{t-q}, is

$$\Delta \alpha_{kj}^{t-q} \propto v_k^{t-q} \left(v_j^t - \langle \mathrm{v}_j^t \rangle_{\mathrm{recon}} \right). \tag{18.95}$$

In the sequel, we introduce the term *temporal pattern* to denote the sequence of feature vectors $\{\boldsymbol{v}^{t-Q}, \boldsymbol{v}^{t-Q+1}, \ldots, \boldsymbol{v}^{t-1}, \boldsymbol{v}^t\}$. After the temporal patterns have been generated from the training set, and before the training stage begins, as it is customary with the contrastive divergence algorithm, they are shuffled and grouped to form mini-batches (usually 100 patterns per mini-batch) to minimize the risk that the training algorithm is trapped in a local minimum of the cost function. The step-size for Eqs. (18.91)–(18.95) needs to be set equal to a small value (e.g., 0.0001). This small value is important to achieve convergence. All weights and biases are initialized with small values using a Gaussian generator.

Remarks 18.7.

- At the time this book is being compiled there is much research activity on the deep learning topic, and it is hard to think of an application area of machine learning in which deep learning has not been applied. I will provide just a few samples of papers, and there is no claim that this covers the whole happening.

  A very successful application of deep networks has been reported in the area of speech recognition, and significant performance improvements have been reported, compared to previously available state-of-the art methods, see, for example, [38]. An application concerning speech-music discrimination is given in [66]. In [94], a visual tracking application is considered. An application on object recognition is reported in [86]. In [63], an application on context-based music recommendation is discussed. The case of large-scale image classification is considered in [81]. These are just a few samples of diverse areas in which deep learning has been applied.

  Some more recent review articles are [11, 18].

18.11 CASE STUDY: A DEEP NETWORK FOR OPTICAL CHARACTER RECOGNITION

The current example demonstrates how a deep neural network can be adopted to classify printed characters. Such classifiers constitute an integral part of what is known as an optical character recognition (OCR) system. For pedagogical purposes, and to keep the system simple, we focus on a four-class scenario (the extension to more classes is straightforward). The characters (classes) that are involved are the Greek letters α, ν, o, and τ, extracted from old historical documents.

Each one of the classes comprises a number of *binarized* images. Each binary image is the result of a segmentation and binarization procedure from scanned documents and all binary images have been reshaped to the same dimensions, that is, 28×28 pixels. This specific dimension has been chosen to comply with the format of the MNIST[6] data set of handwritten digits. Note that, as is common with real-world problems, due to segmentation and binarization errors, several images contain noisy or partially complete characters.

In our example, the class volumes are 1735, 1850, 2391, and 2264 images for classes α, ν, o, and τ, respectively. Examples of these characters can be seen in Figure 18.17. The complete data set can be downloaded from the companion site of this book.

Each binary image is converted to a binary feature vector by scanning it row-wise and concatenating the rows to form a $28 \times 28 = 784$-dimensional binary representation. In the sequel, 80% of the resulting patterns, per class, are randomly chosen to form the training set and the remaining patterns serve testing purposes. The class labels are represented by 4-digit binary codewords. For example, the first class (letter α) is assigned the binary code (1000), the second class is assigned the codeword (0100), and so on.

Because of the binary nature of the patterns, the use of RBMs with binary stochastic units as the building blocks of a deep network is a natural choice. In our case, the adopted deep architecture follows closely the block diagram of Figure 18.13, and consists of five layers in total: an input layer, $\mathbf{x}$, of 784 binary visible units, three layers, namely $\mathbf{h}^1$, $\mathbf{h}^2$, and $\mathbf{h}^3$, of hidden binary units (consisting of 500, 500, and 2000 nodes respectively) and, finally, an output layer, $\mathbf{y}$, of four softmax units, which provide the posterior probability estimates of the patterns for each one of the classes.

Our main goal is to pre-train the weights of this network, via the contrastive divergence (CD) algorithm and use the resulting weight values to initialize the backpropagation algorithm; the latter will eventually provide a fine-tuning of the network's weights. As has been explained in the text, we proceed in a layer-wise mode. We are going to repeat some of the comments made before: "repetitio est mater studiorum."[7]

FIGURE 18.17

Examples of the letters α, ν, o, and τ of the data set under study.

[6] http://yann.lecun.com/exdb/mnist/.

[7] Repetition is mother of study, in Latin.

- We treat nodes $(\mathbf{x}, \mathbf{h}^1)$ as an RBM and use the CD Algorithm in (18.4) for 50 epochs to compute the weights θ_{ij}^1, $i = 1, \ldots, 500$, $j = 1, \ldots, 784$, and biases, b_i^1, $i = 1, \ldots, 500$, of the hidden nodes, and c_j , $j = 1, 2, \ldots, 784$.
- After the first RBM has been trained, we compute the activation outputs of the nodes in layer $\mathbf{h}^1$, for all the patterns in the training set and use the respective activation probabilities as the visible input data of the second RBM, $(\mathbf{h}^1, \mathbf{h}^2)$. As an alternative, we could binarize the activation outputs of the nodes in $\mathbf{h}^1$ to use binary data as inputs in the second RBM; however, in practice, using binarization tends to yield inferior performance. After the second RBM has been trained, the values of weights θ_{ij}^2, $i = 1, \ldots, 500$, $j = 1, \ldots, 500$ and hidden biases, b_i^2, become available.
- We proceed in a manner similar to the third RBM, $(\mathbf{h}^2, \mathbf{h}^3)$, whose weights and hidden node biases are denoted as θ_{ij}^3, $i = 1, \ldots, 2000$, $j = 1, \ldots, 500$, and b_i^3, $i = 1, \ldots, 2000$, respectively. This time, the activation outputs of the nodes in $\mathbf{h}^2$ become the visible input data of the third RBM.
- In the previous stages, three RBMs were trained in total for the first four layers of the network ($\boldsymbol{x}$, $\boldsymbol{h}^i$, $i = 1, \ldots, 3$). In all three training stages, the respective training procedure was unsupervised in the sense that we did not use the information regarding the class labels. The class labels are used for the first time for the supervised training of the weights, θ_{ij}^4, $i = 1, \ldots, 4$, $j = 1, \ldots, 2000$, connecting layer $\mathbf{h}^3$ with the softmax nodes (Eq. 18.41), associated with the output nodes, $\mathbf{y}$. Specifically, we use the backpropagation algorithm to pre-train the θ_{ij}^4's, using the activation outputs $(\mathbf{h}^3)$ of the third RBM for each one of the training patterns, as the input vectors and the codeword of the respective class label as the desired output. Although this type of backpropagation procedure lasts for only a few epochs (10 epochs in our experiment), it helps the initialization of the weights in $\boldsymbol{\theta}^4$, for the backpropagation in the final fine-tuning stage, which will follow; otherwise, initialization should be performed randomly.
- After $\boldsymbol{\theta}^4$ has been pre-trained, a standard backpropagation algorithm that minimizes the cross-entropy cost function (Section 18.4.3) is employed, using fifty epochs to fine-tune all network weights and biases.

During the testing stage, each unknown pattern is "clamped" on the visible nodes of the input layer, $\mathbf{x}$, and the network operates in a feed-forward mode to propagate the results until the output layer, $\mathbf{y}$, has been reached. During this feed-forward operation, the nodes of the hidden layers propagate activation outputs, that is, the probabilities at the output of their logistic functions. Also, note that each output (softmax) node, y_i, $i = 1, \ldots, 4$, emits normalized values in the range $[0, 1]$ and $\sum_{i=1}^{4} y_i = 1$; this allows the interpretation of the corresponding values as posterior probabilities. For each input pattern, the softmax node corresponding to the maximum value is chosen as the winner, and the pattern is assigned to the respective class. For example, if node y_2 wins, the pattern that was "clamped" in the input layer is assigned to class 2 (letter ν).

Figure 18.18 presents the training and testing error curves at the end of each training epoch. Note that due to the small number of classes and network size, the resulting errors become really small after just a few epochs. In this case, the errors are mainly due to seriously distorted characters. Furthermore, observe that the training error (as a general trend) keeps decreasing, while the test error reaches a minimum level, which can be interpreted as an indication that, in this case, we have avoided overfitting. Note, however, that this does not mean that deep networks are free from overfitting problems, in general;

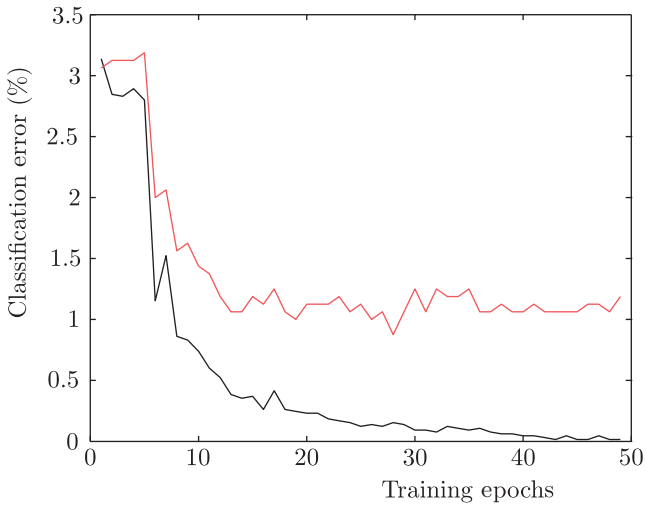

FIGURE 18.18

Training error (gray) and testing error (red) versus number of epochs for the data set of case study (Section 18.11).

see, for example [83], and the references therein. Observe that the probability of error, after convergence, settles close to 1%. This experimental setup can be readily extended to cover more classes, such as, all the letters of the Greek or Latin alphabets.

18.12 CASE STUDY: A DEEP AUTOENCODER

In Section 18.10, we gave the definition of an autoencoder and discussed the use of autoencoders as building blocks for designing deep networks, as an alternative to using RBMs. Now we turn our attention to the use of RBS in designing deep autoencoders for dimensionality reduction and data compression. The idea was first proposed in [35].

The goal of the encoder is to gradually reduce the dimensionality of the input vectors, which will be achieved by using a multilayer neural network, where the hidden layers decrease in size [35]. We demonstrate the method via an example, using the database of the Greek letters discussed before and following the same procedure concerning the partition in training and test data.

Figure 18.19 shows the block diagram of the encoder. It comprises four hidden layers, $\mathbf{h}^i, i = 1, \ldots, 4$, with 1000, 500, 250, and 30 hidden nodes, respectively. The first three hidden layers consist of binary units, whereas the last layer consists of linear (Gaussian) units. We then proceed by pre-training the weights connecting every pair of successive layers using the contrastive divergence algorithm (for 20 epochs), starting from $(\mathbf{x}, \mathbf{h}^1)$ and proceeding with $(\mathbf{h}^1, \mathbf{h}^2)$, and so on. This is in

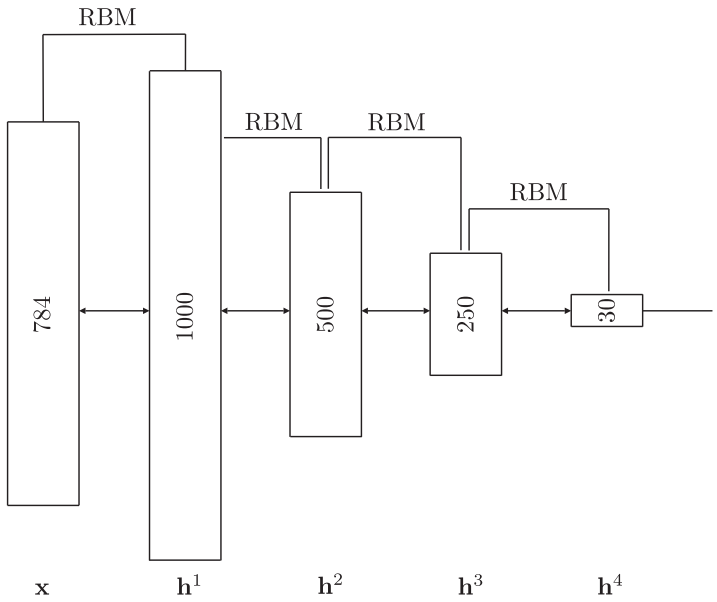

FIGURE 18.19

The block diagram for the encoder.

line with what we discussed so far for training deep networks. For the RBM training stage, the whole training data set was used and divided into mini-batches (consisting of 100 patterns), as is common practice.

The decoder is the reverse structure, that is, its input layer receives the 30-dimensional representation at the output of $\mathbf{h}^4$ and consists of four hidden layers of increasing size, whose dimensions reflect exactly the hidden layers of the encoder, plus an output layer. This is shown in Figure 18.20. It is important to note that the weights of the decoder are not pre-initialized separately; we employ the transpose of the respective weights of the encoder. For example, the weights connecting $\mathbf{h}^5$ with $\mathbf{h}^6$ are initialized with $\boldsymbol{\Theta}_{h_3h_4}^T$, where $\boldsymbol{\Theta}_{h_3h_4}$ are the weights connecting layers $\mathbf{h}^3$ and $\mathbf{h}^4$ of the encoder. The layer denoted as $\mathbf{x}_{rec}$ is the output layer, which provides the reconstructed version of the input.

After all the weights have been initialized as previously described, the whole encoder-decoder network is treated as a multilayer feed-forward network and the weights are fine-tuned via the backpropagation algorithm (for 200 epochs); for each input pattern, the desired output is the pattern itself. In this way, the backpropagation algorithm tries to minimize the reconstruction error. During the backpropagation training procedure, ten mini-batches are grouped together to form a larger batch, and the weights are updated at the end of the processing of each one of these batches. This is a recipe that has proven to provide better convergence in practice.

Figure 18.21 presents some of the patterns of the training set along with their reconstructions before the fine-tuning stage (backpropagation training). Similarly, Figure 18.22 presents the reconstruction results for the same patterns after the fine-tuning stage has been completed. It can be

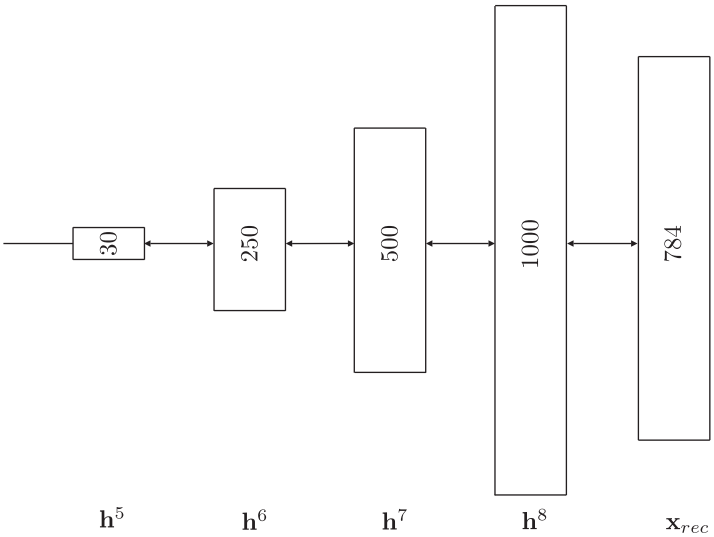

FIGURE 18.20

The block diagram for the decoder.

FIGURE 18.21

Input patterns and respective reconstructions. The top row shows the original patterns. The bottom row shows the corresponding reconstructed patterns, prior to the application of the backpropagation algorithm for fine-tuning.

FIGURE 18.22

Input patterns and respective reconstructions. The top row shows the original patterns. The bottom row shows the corresponding reconstructed patterns after the fine-tuning stage.

readily observed that the application of the backpropagation algorithm yields improved (less noisy) reconstructions.

Finally, Figure 18.23 presents the mean-square reconstruction error (MSE) over all pixels of the images of the training set (black curve) and the testing set (red curve). The MSE is computed during the fine-tuning stage in the beginning of each epoch.

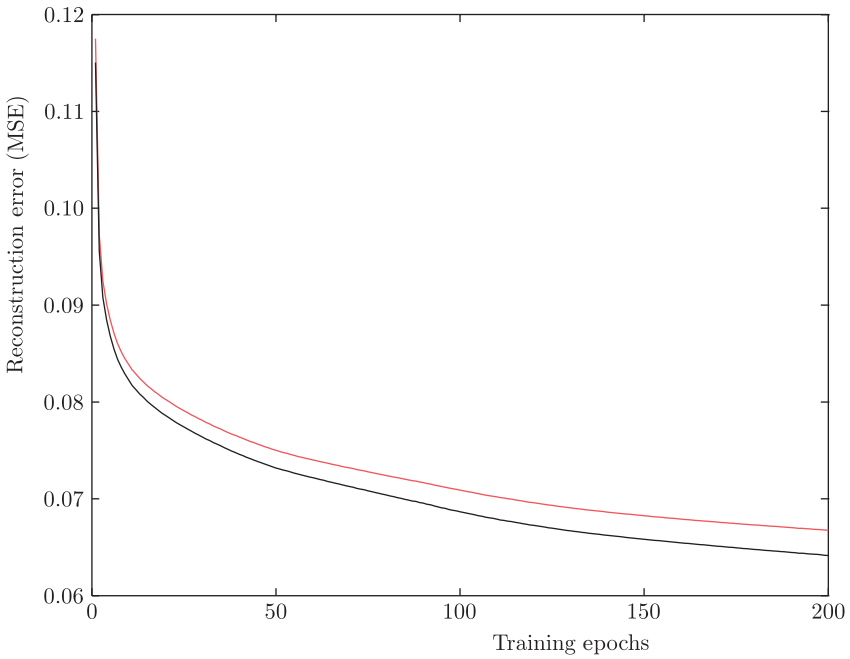

FIGURE 18.23

Mean-square error during the fine-tuning stage, using the backpropagation algorithm for the case study in Section 18.12.

18.13 EXAMPLE: GENERATING DATA VIA A DBN

The current example demonstrates the potential of a deep belief network (DBN) to generate data. Our example evolves around the previously introduced data set of the Greek letters α, ν, o and τ.

The proposed DBN follows the architecture of Figure 18.15b, where $\mathbf{h}^1$, $\mathbf{h}^2$ and $\mathbf{h}^3$ contain 500, 500, and 2000 nodes respectively.

To generate the samples, we follow the steps of Algorithm 18.6. To speed up data generation, each time, we feed a pattern of the data set to the input layer and we propagate the results until layer $\mathbf{h}^2$. The activation probabilities of this layer serve to initialize an alternating Gibbs sampling procedure that runs for 5000 iterations. After the sampling chain has been completed, we perform a single down-pass to generate the data at the input layer.

Figure 18.24 presents the data generation results (even rows), along with the patterns that were used to initialize the procedure each time (odd rows). For the sake of clarity of presentation, the generated data are the activation outputs of the visible layer, that is, we do not binarize the final data generation step.

FIGURE 18.24

Generated data via a DBN. Odd rows show the data used in the input and even rows show the corresponding data generated by the network.

PROBLEMS

18.1 Prove that the perceptron algorithm, in its pattern-by-pattern mode of operation, converges in a finite number of iteration steps. Assume that $\boldsymbol{\theta}^{(0)} = \mathbf{0}$.
Hint. Note that because classes are assumed to be linearly separable, there is a normalized hyperplane, $\boldsymbol{\theta}_*$, and a $\gamma > 0$, so that

$$\gamma \leq y_n \boldsymbol{\theta}_*^T \boldsymbol{x}_n, \quad n = 1, 2, \ldots, N,$$

where, y_n is the respective label, being $+1$ for ω_1 and -1 for ω_2. By the term *normalized hyperplane*, we mean that,

$$\boldsymbol{\theta}_*^T = [\hat{\boldsymbol{\theta}}_*, \theta_{0*}]^T, \text{ with } ||\hat{\boldsymbol{\theta}}_*|| = 1.$$

In this case, $y_n \boldsymbol{\theta}_*^T \boldsymbol{x}_n$ is the distance of $\boldsymbol{x}_n$ from the hyperplane $\boldsymbol{\theta}_*$ ([61]).

18.2 The derivative of the sigmoid functions has been computed in Problem 7.6. Compute the derivative of the hyperbolic tangent function and show that it is equal to,

$$f'(z) = ac\big(1 - f^2(z)\big).$$

18.3 Show that the effect of the momentum term in the gradient descent backpropagation scheme is to effectively increase the learning convergence rate of the algorithm.
Hint. Assume that the gradient is approximately constant over I successive iterations.

18.4 Show that if (a) the activation function is the hyperbolic tangent (b) the input variables are normalized to zero mean and unit variance, then to guarantee that all the outputs of the neurons are zero mean and unit variance, the weights must be drawn from a distribution of zero mean and standard deviation equal to

$$\sigma = m^{-1/2},$$

where m is the number of synaptic weights associated with the corresponding neuron.
Hint. For simplicity, consider the bias to be zero, and also that the inputs to each neuron are mutually uncorrelated.

18.5 Consider the sum of error squares cost function

$$J = \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{k_L}(\hat{y}_{nm} - y_{nm})^2. \tag{18.96}$$

Compute the elements of the Hessian matrix

$$\frac{\partial^2 J}{\partial\theta_{kj}^{r}\partial\theta_{k'j'}^{r'}}. \tag{18.97}$$

Near the optimum, show that the second order derivatives can be approximated by

$$\frac{\partial^2 J}{\partial\theta_{kj}^{r}\partial\theta_{k'j'}^{r'}} = \sum_{n=1}^{N}\sum_{m=1}^{k_L}\frac{\partial\hat{y}_{nm}}{\partial\theta_{kj}^{r}}\frac{\partial\hat{y}_{nm}}{\partial\theta_{k'j'}^{r'}}. \tag{18.98}$$

In other words, the second order derivatives can be approximated as products of the first order derivatives. The derivatives can be computed by following similar arguments as the gradient descent backpropagation scheme [25].

18.6 It is common when computing the Hessian matrix to assume that it is diagonal. Show that under this assumption, the quantities

$$\frac{\partial^2 E}{\partial(\theta_{kj})^2},$$

where

$$E = \sum_{m=1}^{k_L}\left(f(z_m^L) - y_m\right)^2,$$

propagates backward according to the following,

- $\dfrac{\partial^2 E}{\partial(\theta_{kj}^{r})^2} = \dfrac{\partial^2 E}{\partial(z_j^r)^2}(y_k^{r-1})^2.$
- $\dfrac{\partial^2 E}{\partial(z_L^r)^2} = f''(z_j^L)e_j + (f'(z_j^L))^2.$
- $\dfrac{\partial^2 E}{\partial(z_k^{r-1})^2} = (f'(z_j^{r-1}))^2\sum_k \dfrac{\partial^2 E}{\partial(z_k^r)^2}(\theta_{kj}^r)^2 + f''(z_j^{r-1})\sum_{k=1}^{k_r}\theta_{kl}^r\delta_k^r.$

18.7 Show that the cross-entropy loss function depends on the relative output errors.

18.8 Show that if the activation function is the logistic sigmoid and the relative entropy cost function is used, then δ_{nj}^L in Eq. (18.24) becomes,

$$\delta_{nj}^L = a(\hat{y}_{nj} - 1)y_{nj}.$$

18.9 As in the previous problem, use the relative entropy cost function and the softmax activation function. Then show that,

$$\delta_{nj}^L = \hat{y}_{nj} - y_{nj}.$$

18.10 Derive the gradient of the log-likelihood in Eq. (18.61).

18.11 Derive the factorization of the conditional probability in Eq. (18.63).

18.12 How are Eqs. (18.81) to (18.83) and the contrastive divergence algorithm modified for the case of an RBM with binary visible and Gaussian hidden nodes?

MATLAB Exercises

18.13 Consider a two-dimensional class problem that involves two classes ω_1 $(+1)$ and ω_2 (-1). Each one of them is modeled by a mixture of equiprobable Gaussian distributions. Specifically, the means of the Gaussians associated with ω_1 are $[-5,\ 5]^T$ and $[5,\ -5]^T$, while the means of the Gaussians associated with ω_2 are $[-5,\ -5]^T$, $[0,\ 0]^T$ and $[5,\ 5]^T$. The covariances of all Gaussians are $\sigma^2 I$, where $\sigma^2 = 1$.

(i) Generate and plot a data set X_1 (training set) containing 100 points from ω_1 (50 points from each associated Gaussian) and 150 points from ω_2 (again 50 points from each associated Gaussian). In the same way, generate an additional set X_2 (test set).

(ii) Based on X_1, train a two-layer neural network with two nodes in the hidden layer having the hyperbolic tangent as activation function and a single output node with linear activation function[8], using the standard backpropagation algorithm for 9000 iterations and step-size equal to 0.01. Compute the training and the test errors, based on X_1 and X_2, respectively. Also, plot the test points as well as the decision lines formed by the network. Finally, plot the training error versus the number of iterations.

(iii) Repeat step (ii) for step-size equal to 0.0001 and comment on the results.

(iv) Repeat step (ii) for $k = 1, 4, 20$ hidden layer nodes and comment on the results.

Hint. Use different seeds in the *rand* MATLAB function for the train and the test sets. To train the neural networks, use the *newff* MATLAB function. To plot the decision region performed by a neural network, first determine the boundaries of the region where the data live (for each dimension determine the minimum and the maximum values of the data points), then apply a rectangular grid on this region and for each point in the grid compute the output of the network. Then draw this point with different colors according to the class it is assigned (use e.g., the "magenta" and the "cyan" colors).

18.14 Consider the classification problem of the previous exercise, as well as the same data sets X_1 and X_2.

Consider a two-layer feed-forward neural network as the one in (ii) and train it using the adaptive backpropagation algorithm with initial step-size equal to 0.0001 and $r_i = 1.05$, $r_d = 0.7$, $c = 1.04$, for 6000 iterations. Compute the training and the test errors, based on X_1 and X_2, respectively and plot the error during training against the number of iterations. Compare the results with those obtained from the previous exercise (ii).

18.15 Repeat the previous exercise for the case where the covariance matrix for the Gaussians is $6I$, for 2, 20 and 50 hidden layer nodes, compute the training and the test errors in each case and draw the corresponding decision regions. Draw your conclusions.

18.16 Develop a MATLAB program which implements the experiment of the case study in Section 18.11, skipping the RBM pre-training stage. You will first need to download the OCR data set from the companion website of this book. Your program will accept as input the

[8] The number of input nodes equals to the dimensionality of the feature space, while the number of output nodes is equal to the number of classes minus one.

number of nodes of each layer of the network. Call MATLAB's backpropagation function to initialize the network weights with random numbers and train directly the network as a whole. During the training stage, plot the training error in the beginning of each training epoch. Do you observe any differences regarding the generated error curve compared with the one in Section 18.11? Provide a justification of your answer.

18.17 Download the OCR data set from the companion website of this book. Then, develop a MATLAB function that receives as input the path to the folder containing the test data set and produces a new data set by corrupting each binary image with noise as follows: 5% of the pixels are randomly chosen from each image and their values are altered, that is, a 0 becomes 1 and vice versa. After the corrupted data set has been generated and stored to a new folder, create a function, such as, deepClassifier.m, that feeds it to the trained network of Section 18.11 and computes the resulting test error. The weights of the trained network are available at the OCRTrained1.mat file. Repeat the error computation by increasing the noise intensity in a stepwise mode, that is, by 1% each time. How is the performance of the deep network affected?

18.18 Repeat the experiment in Exercise 18.17, using binary outputs for the hidden nodes in all stages, instead of activation probabilities. This holds for the pre-training stages and the feed-forward classification procedure. Do you observe any performance deterioration?

18.19 Develop the MATLAB code and repeat the autoencoder experiment in 18.12. Corrupt the data set (both training and test sets) with noise by randomly altering the values of 5% of the pixels of each image and repeat the training procedure. Plot the reconstructed input and MSE curves and comment on the results.

REFERENCES

[1] D. Ackle, G.E. Hinton, T. Sejnowski, A learning algorithm for Boltzmann machines, Cognit. Sci. 9 (1985) 147-169.

[2] T. Adali, X. Liu, K. Sonmez, Conditional distribution learning with neural networks and its application to channel equalization, IEEE Trans. Signal Process. 45 (4) (1997) 1051-1064.

[3] P. Baldi, K. Hornik, Neural networks and principal component analysis: learning from examples, without local minima, Neural Netw. 2 (1989) 53-58.

[4] E. Barnard, Optimization for training neural networks, IEEE Trans. Neural Netw. 3 (2) (1992) 232-240.

[5] R.A. Barron, Universal approximation bounds for superposition of a sigmoidal function, IEEE Trans. Inform. Theory 39 (3) (1993) 930-945.

[6] R. Battiti, First and second order methods for learning: Between steepest descent and Newton's methods, Neural Comput. 4 (1992) 141-166.

[7] Y. Bengio, O. Delalleau, N. Le Roux, The curse of highly variable functions for local kernel machines, in: Y. Weiss, B. Schölkopf, J. Platt (Eds.), Advances in Neural Information Processing Systems (NIPS), vol. 18, MIT Press, Cambridge, MA, 2006, pp. 107-114.

[8] Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle, Greedy layer-wise training of deep networks, in: B. Schölkopf, J. Platt, T. Hofmann (Eds.), Advances in Neural Information Processing Systems (NIPS), vol. 19, MIT Press, Cambridge, MA, 2007, pp. 153-161.

[9] Y. Bengio, Learning deep architectures for AI, Found. Trends Mach. Learn. 2 (1) (2009) 1-127, DOI: 10.1561/2200000006.

[10] Y. Bengio, O. Delalleau, Justifying and generalizing contrastive divergence, Neural Comput. 21 (6) (2009) 1601-1621.
[11] Y. Bengio, A. Courville, P. Vincent, Unsupervised feature learning and deep learning: a review and new perspectives, 2014, arXiv:1206.5538v3 [cs.LG] 23 April 2014.
[12] C.M. Bishop, Neural Networks for Pattern Recognition, Oxford University Press, Oxford, 1995.
[13] J.S. Bridle, Training stochastic model recognition algorithms as networks can lead to maximum information estimation parameters, in: D.S. Touretzky, et al. (Eds.), Neural Information Processing Systems, NIPS, vol. 2, Morgan Kaufmann, San Francisco, CA, 1990, pp. 211-217.
[14] A. Bryson, W. Denham, S. Dreyfus, Optimal programming problems with inequality constraints I: Necessary conditions for extremal solutions, J. Am. Inst. Aeronaut. Astronaut. 1 (1963) 25-44.
[15] M. Carreira-Perpinan, G.E. Hinton, On contrastive divergence learning, in: Proceedings 10th International Workshop on Artificial Intelligence and Statistics (AISTATS), 2005, pp. 59-66.
[16] A. Cichoki, R. Unbenhauen, Neural Networks for Optimization and Signal Processing, John Wiley, New York, 1993.
[17] G. Cybenko, Approximation by superpositions of a sigmoidal function, Math. Control Signals Syst. 2 (1989) 304-314.
[18] L. Deng, Y. Dong, Deep Learning: Methods and Applications, vol. 7(3-4), 2014.
[19] D. Erhan, P.A. Manzagol, Y. Bengio, S. Bengio, P. Vincent, The difficulty of training deep architectures and the effect of unsupervised pretraining, in: Proceedings of The Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS09), 2009, pp. 153-160.
[20] S.E. Fahlman, Faster learning variations on back-propagation: an empirical study, in: Proceedings Connectionist Models Summer School, Morgan Kaufmann, San Francisco, CA, 1988, pp. 38-51.
[21] Y. Freund, D. Haussler, Unsupervised learning of distributions of binary vectors using two layer networks, Technical Report, UCSC-CRL-94-25, 1994.
[22] K. Fukushima, Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position, Biol. Cybern. 36 (1980) 193-202.
[23] K. Funashashi, On the approximation realization of continuous mappings by neural networks, Neural Netw. 2 (3) (1989) 183-192.
[24] M. Hagiwara, Theoretical derivation of momentum term in backpropagation, in: International Joint Conference on Neural Networks, Baltimore, vol. I, 1991, pp. 682-686.
[25] B. Hassibi, D.G. Stork, G.J. Wolff, Optimal brain surgeon and general network pruning, in: Proceedings IEEE Conference on Neural Networks, vol. 1, 1993, pp. 293-299.
[26] S. Haykin, Neural Networks, second ed., Prentice Hall, Upper Saddle River, NJ, 1999.
[27] D.O. Hebb, The Organization of Behavior: A Neuropsychological Theory, Wiley, New York, 1949.
[28] J. Hertz, A. Krogh, R.G. Palmer, Introduction to the Theory of Neural Computation, Addison-Wesley, Reading, MA, 1991.
[29] G.E. Hinton, T.J. Sejnowski, Optimal perceptual inference, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Washington, DC, June, 1983.
[30] G.E. Hinton, Learning distributed representations of concepts, in: Proceedings of the Eighth Annual Conference of the Cognitive Science Society, Amherst, Lawrence Erlbaum, Hillsdale, 1986, pp. 1-12.
[31] G.E. Hinton, T.J. Sejnowski, Learning and relearning in Boltzmann machines, in: D.E. Rumelhart, J.L. McClelland (Eds.), Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol. 1, MIT Press, Cambridge, MA, 1986, pp. 282-317.
[32] G.E. Hinton, P. Dayan, B.J. Frey, R.M. Neal, The wake-sleep algorithm for unsupervised neural networks, Science 268 (1995) 1558-1161.
[33] G.E. Hinton, Training products of experts by minimizing contrastive divergence, Neural Comput. 14 (2002) 1771-1800.

[34] G.E. Hinton, S. Osindero, Y. Teh, A fast learning algorithm for deep belief nets, Neural Comput. 18 (2006) 1527-1554.
[35] G.E. Hinton, R. Salakhutdinov, Reducing the dimensionality of data with neural networks, Science 313 (2006) 504-507.
[36] G. Hinton, A Practical Guide to Training Restricted Boltzmann Machines, Technical Report, UTML TR 2010-003, University of Toronto, 2010, http://learning.cs.toronto.edu.
[37] G.E. Hinton, Learning multiple layers of representation, Trends Cognit. Sci. 11 (10) (2010) 428-434.
[38] G. Hinton, L. Deng, D. Yu, G.E. Dahl, A.R. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T.N. Sainath, B. Kinsbury, Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups, IEEE Signal Process. Mag. 29 (6) (2012) 82-97.
[39] K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, Neural Netw. 2 (5) (1989) 359-366.
[40] G.-B. Huang, Q.-Y. Zhu, C.-K. Siew, Extreme learning machine: theory and applications, Neurocomputing 70 (2006) 489-501.
[41] G.B. Huang, L. Chen, C.K. Siew, Universal approximation using incremental constructive feedforward networks with random hidden nodes, IEEE Trans. Neural Netw. 17 (4) (2006) 879-892.
[42] D.H. Hubel, T.N. Wiesel, Receptive fields, binocular interaction, and functional architecture in the cats visual cortex, J. Physiol. 160 (1962) 106-154.
[43] A. Hyvarinen, Connections between score matching, contrastive divergence, and pseudolikelihood for continuous-valued variables, IEEE Trans. Neural Netw. 18 (5) (2007) 1529-1531.
[44] G.-B. Huang, D.-H. Wang, Y. Lan, Extreme learning machines: a survey, Int. J. Mach. Learn. Cybern. 2 (2011) 107-122.
[45] Y. Ito, Representation of functions by superpositions of a step or sigmoid function and their application to neural networks theory, Neural Netw. 4 (3) (1991) 385-394.
[46] R.A. Jacobs, Increased rates of convergence through learning rate adaptation, Neural Netw. 2 (1988) 359-366.
[47] E.M. Johanson, F.U. Dowla, D.M. Goodman, Backpropagation learning for multilayer feedforward neural networks using conjugate gradient method, Int. J. Neural Syst. 2 (4) (1992) 291-301.
[48] A.H. Kramer, A. Sangiovanni-Vincentelli, Efficient parallel learning algorithms for neural networks, in: D.S. Touretzky (Ed.), Advances in Neural Information Processing Systems 1, NIPS, Morgan Kaufmann, San Francisco, CA, 1989, pp. 40-48.
[49] Y. LeCun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard, L.D. Jackel, Backpropagation applied to handwritten zip code recognition, Neural Comput. 1 (4) (1989) 541-551.
[50] Y. LeCun, J.S. Denker, S.A. Solla, Optimal brain damage, in: D.S. Touretzky (Ed.), Advances in Neural Information Systems, vol. 2, Morgan Kaufmann, San Francisco, CA, 1990, pp. 598-605.
[51] Y. LeCun, L. Bottou, G.B. Orr, K.R. Müller, Efficient BackProp, in: G.B. Orr, K.-R. Müller (Eds.), Neural Networks: Tricks of the Trade, Springer, New York, 1998, pp. 9-50.
[52] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proc. IEEE 86 (11) (1998) 2278-2324.
[53] T.S. Lee, D. Mumford, Hierarchical Bayesian inference in the visual cortex, J. Opt. Soc. Am. A 20 (7) (2003) 1434-1448.
[54] T.S. Lee, D.B. Mumford, R. Romero, V.A.F. Lamme, The Role of the Primary Visual Cortex in Higher Level Vision, Vis. Res. 38 (1998) 2429-2454.
[55] N. Le Roux, Y. Bengio "Representational power of restricted boltzmann machines and deep belief networks," Neural Computation, vol. 20(6), pp. 1631–1649, 2008.
[56] D.J.C. MacKay, A practical Bayesian framework for back-propagation networks, Neural Comput. 4 (3) (1992) 448-472.

[57] D.J.C. MacKay, The evidence framework applied to classification networks, Neural Comput. 4 (5) (1992) 720-736.
[58] T.K. Marks, J.R. Movellan, Diffusion networks, product of experts, and factor analysis, in: Proceedings International Conference on Independent Component Analysis, 2001, pp. 481-485.
[59] W. McCulloch, W. Pitts, A logical calculus of ideas immanent in nervous activity, Bull. Math. Biophys. 5 (1943) 115-133.
[60] D.B. Mumford, On the computational architecture of the neocortex. II. The role of cortico-cortical loops, Biol. Cybern. 66 (1992) 241-251.
[61] A.B. Navikoff, On convergence proofs on perceptrons, in: Symposium on the Mathematical Theory of Automata, vol. 12, Polytechnic Institute of Brooklyn, Brooklyn, 1962, pp. 615-622.
[62] R. Neal, Connectionist learning of belief networks, Artif. Intell. 56 (1992) 71-113.
[63] A. van den Oord, S. Dieleman, B. Schrauwen, Deep content-based music recommendation, in: Proceedings Neural Information Processing Systems, 2013.
[64] P. Orponen, Computational complexity of neural networks: A survey, Nordic J. Comput. 1 (1) (1994) 94-110.
[65] S.J. Perantonis, P.J.G. Lisboa, Translation, rotation, and scale invariant pattern recognition by high-order neural networks and moment classifiers, IEEE Trans. Neural Netw. 3 (2) (1992) 241-251.
[66] A. Pikrakis, S. Theodoridis, Speech-music discrimination: A deep learning perspective, in: Proceedings of the 22nd European Signal Processing Conference (EUSIPCO), 1-5 September 2014, Lisbon, Portugal, 2014.
[67] R. Rajesh, J.S. Prakash, Extreme learning machines—A review and state-of-the-art, Int. J. Wisdom Based Comput. 1 (1) (2011) 35-49.
[68] S. Ramon y Cajal, Histologia du Systéms Nerveux de l' Homme et des Vertebes, vols. I, II, Maloine, Paris, 1911.
[69] L.P. Ricotti, S. Ragazzini, G. Martinelli, Learning the word stress in a suboptimal second order backpropagation neural network, in: Proceedings IEEE International Conference on Neural Networks, San Diego, vol. 1, 1988, pp. 355-361.
[70] M. Riedmiller, H. Brau, A direct adaptive method for faster backpropagation learning: the prop algorithm, in: Proceedings of the IEEE Conference on Neural Networks, San Francisco, 1993.
[71] S. Rifai, P. Vincent, X. Muller, X. Gloro, Y. Bengio, Contractive auto-encoders: explicit invariance during feature extraction, in: Proceedings of the 28th International Conference on Machine Learning (ICML), Bellevue, WA, USA, 2011.
[72] J. Rissanen, G.G. Langdon, Arithmetic coding, IBM J. Res. Dev. 23 (1979) 149-162.
[73] F. Rosenblatt, The perceptron: A probabilistic model for information storage and organization in the brain, Psychol. Rev. 65 (1958) 386-408.
[74] F. Rosenlatt, Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms, Spartan, Washington, DC, 1962.
[75] D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning representations by backpropagating errors, Nature 323 (1986) 533-536.
[76] R. Russel, Pruning algorithms: a survey, IEEE Trans. Neural Netw. 4 (5) (1993) 740-747.
[77] T. Serre, G. Kreiman, M. Kouh, C. Cadieu, U. Knoblich, T. Poggio, A quantitative theory of immediate visual recognition, in: Progress in Brain Research, Computational Neuroscience: Theoretical Insights into Brain Function, vol. 165, 2007, pp. 33-56.
[78] J. Sietsma, R.J.F. Dow, Creating artificial neural networks that generalize, Neural Netw. 4 (1991) 67-79.
[79] E.M. Silva, L.B. Almeida, Acceleration techniques for the backpropagation algorithm, in: L.B. Almeida, et al. (Eds.) Proceedings on the EURASIP Workshop on Neural Networks, Portugal, 1990, pp. 110-119.
[80] P.Y. Simard, D. Steinkraus, J. Platt, Best practice for convolutional neural networks applied to visual document analysis, in: Proceedings International Conference on Document Analysis and Recognition, ICDAR, 2003, pp. 958-962.

[81] K. Simonyan, A. Vedaldi, A. Zisserman, Deep Fisher networks for large-scale image classification, in: Proceedings Neural Information processing Systems, 2013.
[82] P. Smolensky, Information processing in dynamical systems: foundations of harmony theory, in: Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol. 1, 1986, pp. 194-281.
[83] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov "Dropout: A simple way to prevent neural networks form overfitting" Journal of Machine Learning Research, Vol. 15, pp. 1929–1958, 2014.
[84] I. Sutskever, T. Tieleman, On the convergence properties of contrastive divergence, in: Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS), vol. 9, Chia Laguna Resort, Sardinia, Italy, 2010.
[85] K. Swersky, B. Chen, B. Marlin, N. Nando de Freitas, A tutorial on stochastic approximation algorithms for training restricted Boltzmann machines and deep belief nets, in: Proceedings of the Information Theory and Applications Workshop (ITA), San Diego, 31 January 2010-5 February 2010, 2010, pp. 1-10.
[86] C. Szegedy, A. Toshev, D. Erhan, Deep neural networks for object detection, in: Proceedings Neural Information Processing Systems, 2013.
[87] G.W. Taylor, G.E. Hinton, S.T. Roweis, Modeling human motion using binary latent variables, in: Advances in Neural Information Processing Systems, 2006, pp. 1345-1352.
[88] S. Theodoridis, K. Koutroumbas, Pattern Recognition, fourth ed., Academic Press, Boston, 2009.
[89] T. Tieleman, Training restricted Boltzmann machines using approximations to the likelihood gradient, in: Proceedings of the 25th International Conference on Machine Learning, ACM, New York, NY, USA, 2008, pp. 1064-1071.
[90] P.E. Utgoff, D.J. Stracuzzi, Many-layered learning, Neural Comput. 14 (2002) 2497-2539.
[91] V.N. Vapnik, The Nature of Statistical Learning Theory, Springer, New York, 1995.
[92] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, P.A. Manzagol, Stacked denoising autoencoders: learning useful representations in a deep network with a local denoising criterion, J. Mach. Learn. Res. 11 (2010) 3371-3408.
[93] P. Vincent, A connection between score matching and denoising autoencoders, Neural Comput. 23 (7) (2011) 1661-1674.
[94] N. Wang, D.-Y. Yeung, Learning a deep compact image representation for visual tracking, in: Proceedings Neural Information processing Systems (NIPS), 2013.
[95] R.L. Watrous, Learning algorithms for connectionist networks: applied gradient methods of nonlinear optimization, in: Proceedings on the IEEE International Conference on Neural Networks, vol. 2, 1988, pp. 619-627.
[96] A.S. Weigend, D.E. Rumerlhart, B.A. Huberman, Backpropagation, weight elimination and time series prediction, in: D. Touretzky, J. Elman, T. Sejnowski, G. Hinton (Eds.), Proceedings, Connectionist Models Summer School, 1990, pp. 105-116.
[97] P.J. Werbos, Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences, PhD Thesis, Harvard University, Cambridge, MA, 1974.
[98] B. Widrow, M.E. Hoff Jr., Adaptive switching networks, IRE WESCON Convention Record, 1960, pp. 96-104.
[99] W. Wiegerinck, A. Komoda, T. Heskes, Stochastic dynamics on learning with momentum in neural networks, J. Phys. A 25 (1994) 4425-4437.
[100] A. Yuille, The convergence of contrastive divergences, in: L.K. Saul, W. Weis, L. Botou (Eds.), Advances in Neural Information Processing Systems (NIPS), vol. 17, 2004, pp. 1593-1601.
[101] L. Younes, Parametric inference for imperfectly observed Gibbsian fields, Probab. Theory Relat. Fields 82 (4) (1989) 625-645.
[102] J. Zourada, Introduction to Artificial Neural Networks, West Publishing Company, St. Paul, MN, 1992.