Chapter 5
Artificial Neural Networks

Polynomial classifiers can model decision surfaces of any shape; and yet their practical utility is limited because of the easiness with which they overfit noisy training data, and because of the sometimes impractically high number of trainable parameters. Much more popular are *artificial neural networks* where many simple units, called *neurons*, are interconnected by weighted links into larger structures of remarkably high performance.

The field of neural networks is too rich to be covered in the space we have at our disposal. We will therefore provide only the basic information about two popular types: multilayer perceptrons and radial-basis function networks. The chapter describes how each of them classifies examples, and then describes some elementary mechanisms to induce them from training data.

5.1 Multilayer Perceptrons as Classifiers

Throughout this chapter, we will suppose that all attributes are continuous. Moreover, it is practical (though not strictly necessary) to assume that they have been normalized so that their values always fall in the interval $[-1, 1]$.

Neurons. The function of a *neuron*, the basic unit of a multilayer perceptron, is quite simple. A weighted sum of signals arriving at the input is subjected to a *transfer function*. Several different transfer functions can be used; the one that is preferred in this chapter is the so-called *sigmoid*, defined by the following formula where Σ is the weighted sum of inputs:

$$f(\Sigma) = \frac{1}{1 + e^{-\Sigma}} \tag{5.1}$$

M. Kubat, *An Introduction to Machine Learning*,
DOI 10.1007/978-3-319-20010-1_5

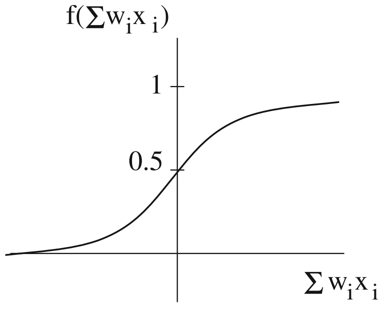


Fig. 5.1 A popular transfer function: the `sigmoid`

Figure 5.1 shows the curve representing the transfer function. The reader can see that $f(\Sigma)$ grows monotonically with the increasing value of Σ but is destined never to leave the open interval $(0, 1)$ because $f(-\infty) = 0$, and $f(\infty) = 1$. The vertical axis is intersected at f(0) = 0.5. We will assume that each neuron has the same transfer function.

Multilayer perceptron. The neural network in Fig. 5.2 is known as the *multilayer perceptron*. The neurons, represented by ovals, are arranged in the *output layer* and the *hidden layer*.[1] For simplicity, we will consider only networks with a single hidden layer while remembering that it is quite common to employ two such layers, even three, though rarely more than that.

While there is no communication between neurons of the same layer, adjacent layers are fully interconnected. Importantly, each neuron-to-neuron link is associated with a *weight*. The weight of the link from the j-th hidden neuron to the i-th output neuron is denoted as $w_{ji}^{(1)}$, and the weight of the link from the k-th attribute to the j-th hidden neuron as $w_{kj}^{(2)}$. Note that the first index always refers to the link's "beginning"; the second, to its "end."

Forward propagation. When we present the network with an example, $\mathbf{x} = (x_1, \ldots, x_n)$, its attribute values are passed along the links to the neurons. The values x_k being multiplied by the weights associated with the links, the j-th hidden neuron receives as input the weighted sum, $\sum_k w_{kj}^{(2)} x_k$, and subjects this sum to the sigmoid, $f(\sum_k w_{kj}^{(2)} x_k)$. The i-th output neuron then receives the weighted sum of the values coming from the hidden neurons and, again, subjects it to the transfer function. This is how the i-th output is obtained. The process of propagating in this manner the attribute values from the network's input to its output is called *forward propagation*.

Each class is assigned its own output neuron, and the value returned by the i-th output neuron is interpreted as the amount of evidence in support of the i-th class. For instance, if the values obtained at three output neurons are $\mathbf{y} = (0.2, 0.6, 0.1)$, the classifier will label the given example with the second class because 0.6 is greater than both 0.2 and 0.1.

[1] When we view the network from above, the hidden layer is obscured by the output layer.

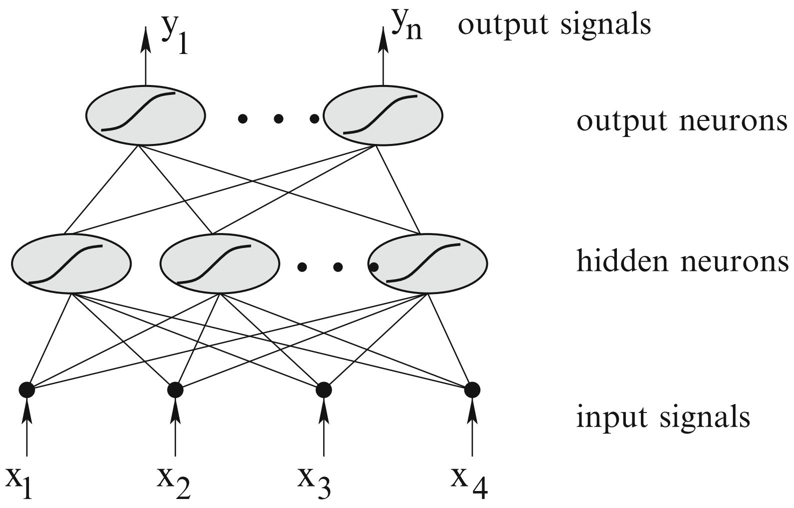


Fig. 5.2 An example neural network consisting of two interconnected layers

In essence, this kind of two-layer perceptron calculates the following formula where f is the sigmoid transfer function (see Eq. 5.1) employed by the neurons; $w_{kj}^{(2)}$ and $w_{ji}^{(1)}$ are the links leading to the hidden and output layers, respectively, and x_k are the attribute values of the presented example:

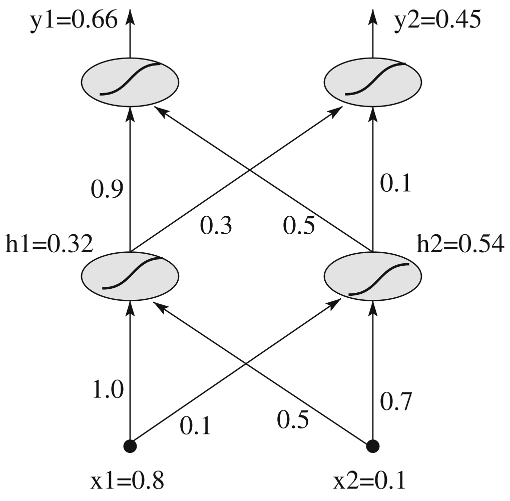
$$y_i = f(\sum_j w_{ji}^{(1)} f(\sum_k w_{kj}^{(2)} x_k)) \tag{5.2}$$

A numeric example. The principle of forward propagation is illustrated by the numeric example in Table 5.1. At the beginning, the attribute vector **x** is presented. Before reaching the neurons in the hidden layer, the attribute values are multiplied by the corresponding weights, and the weighted sums are passed on to the sigmoid functions. The results ($h_1 = 0.32$ and $h_2 = 0.54$) are then multiplied by the next layer of weights, and forwarded to the output neurons where they are again subjected to the sigmoid function. This is how the two output values, $y_1 = 0.66$ and $y_2 = 0.45$, have been obtained. The evidence supporting the class of the "left" output neuron is higher than the evidence supporting the class of the "right" output neuron. The classifier therefore chooses the left neuron's class.

Universal classifier. Mathematicians have been able to prove that, with the right choice of weights, and with the right number of the hidden neurons, Eq. 5.2 can approximate with arbitrary accuracy any realistic function. The consequence of this so-called *universality theorem* is that the *multilayer perceptron* can in principle be used to address just about any classification problem. What the theorem *does not* tell us, though, is how many hidden neurons are needed, and what the individual weight values should be. In other words, we know that the solution exists, yet there is no guarantee we will ever find it.

Table 5.1 Example of forward propagation in a *multilayer perceptron*

Task. Forward-propagate $\mathbf{x} = (x_1, x_2) = (0.8, 0.1)$ through the network below.



Solution.

inputs of hidden-layer neurons:

$$z_1^{(2)} = 0.8 \times (-1.0) + 0.1 \times 0.5 = -0.75$$
$$z_2^{(2)} = 0.8 \times 0.1 + 0.1 \times 0.7 = 0.15$$

outputs of hidden-layer neurons:

$$h_1 = f(z_1^{(2)}) = \frac{1}{1+e^{-(-0.75)}} = 0.32$$
$$h_2 = f(z_2^{(2)}) = \frac{1}{1+e^{-0.15}} = 0.54$$

inputs of output-layer neurons:

$$z_1^{(1)} = 0.54 \times 0.9 + 0.32 \times 0.5 = 0.65$$
$$z_2^{(1)} = 0.54 \times (-0.3) + 0.32 \times (-0.1) = -0.19$$

outputs of output-layer neurons:

$$y_1 = f(z_1^{(1)}) = \frac{1}{1+e^{-0.65}} = 0.66$$
$$y_2 = f(z_2^{(1)}) = \frac{1}{1+e^{-(-0.19)}} = 0.45$$

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain how the example described by a vector of continuous attributes is forward-propagated through the *multilayer perceptron*. How is the network's output interpreted?
- What is the *transfer function*? Write down the formula defining the *sigmoid* transfer function, and describe its shape.
- What is the *universality theorem*? What does it tell us, and what does it *not* tell us?

5.2 Neural Network's Error

Let us defer to a later chapter the explanation of a technique to train the *multilayer perceptron* (to find its weights). Before we can address this issue, it is necessary to prepare the soil by taking a closer look at the method of classification, and at the way the accuracy of this classification is evaluated.

Error rate. Suppose a training example, $\mathbf{x}$, with known class, $c(\mathbf{x})$, has been presented to an existing version of the *multilayer perceptron*. The forward-propagation step establishes the label, $h(\mathbf{x})$. If $h(\mathbf{x}) \neq c(\mathbf{x})$, an error has been made. This may happen to some other examples, too, and we want to know *how often* this happens. We want to know the *error rate*—which is for a given set of examples obtained by dividing the number of errors by the number of examples. For instance, if the classifier misclassifies 30 out of 200 examples, the error rate is $30/200 = 0.15$.

This, however, fails to give the full picture of the network's classification performance. What the error rate neglects to reflect is the sigmoid function's ability to measure the *size* of each error.

An example will clarify the point. Suppose we have two different networks to choose from, each with three output neurons corresponding to classes denoted by C_1, C_2, and C_3. Let us assume that, for some example $\mathbf{x}$, the first network outputs $\mathbf{y}1(\mathbf{x}) = (0.5, 0.2, 0.9)$ and the second, $\mathbf{y}2(\mathbf{x}) = (0.6, 0.6, 0.7)$. This means that both will label $\mathbf{x}$ with the third class, $h1(\mathbf{x}) = h2(\mathbf{x}) = C_3$. If the correct answer is $c(\mathbf{x}) = C_2$, both have erred, but the error does not appear to be the same. The reader will have noticed that the first network was "very sure" about the class being C_3 (because 0.9 is clearly greater than the other two outputs, 0.5 and 0.2), whereas the second network was less certain, the differences of the output values (0.6, 0.6, and 0.7) being so small as to give rise to the suspicion that C_3 has won merely by chance. Due to its weaker commitment to the incorrect class, the second network is somehow less wrong than the first.

This is the circumstance that can be captured by a more appropriate error function, the *mean square error (MSE)*.

Target vector. Before proceeding to the definition of the mean square error, however, we must introduce yet another important concept, the *target vector* which, too, depends on the concrete example, $\mathbf{x}$. Let us denote it by $\mathbf{t}(\mathbf{x})$. In a domain with m classes, the target vector, $\mathbf{t}(\mathbf{x}) = (t_1(\mathbf{x}), \ldots, t_m(\mathbf{x}))$, consists of m binary numbers.

If the example belongs to the i-th class, then $t_i(\mathbf{x}) = 1$ and all other elements in this vector are $t_j(\mathbf{x}) = 0$ (where $j \neq i$). For instance, suppose the existence of three different classes, C_1, C_2, and C_3, and let $\mathbf{x}$ be known to belong to C_2. In the ideal case, the second neuron should output 1, and the two other neurons should output 0.[2] The target is therefore $\mathbf{t}(\mathbf{x}) = (t_1, t_2, t_3) = (0, 1, 0)$.

Mean square error. The mean square error is defined using the differences between the elements of the output vector and the target vector:

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(t_i - y_i)^2 \tag{5.3}$$

When calculating the network's *MSE*, we have to establish for each output neuron the difference between its output and the corresponding element of the target vector. Note that the terms in the parentheses, $(t_i - y_i)$, are squared to make sure that negative differences are not subtracted from positive ones.

Returning to the example of the two networks mentioned above, if the target vector is $\mathbf{t}(\mathbf{x}) = (0, 1, 0)$, then these are the mean square errors:

$$MSE_1 = \tfrac{1}{3}[(0-0.5)^2 + (1-0.2)^2 + (0-0.9)^2] = 0.57$$
$$MSE_2 = \tfrac{1}{3}[(0-0.6)^2 + (1-0.6)^2 + (0-0.7)^2] = 0.34$$

As expected, $MSE_2 < MSE_1$, which is in line with our intuition that the second network is "less wrong" on $\mathbf{x}$ than the first network.

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- In what sense do we say that the traditional error rate does not provide enough information about a neural network's classification accuracy?
- Explain the difference between a neural network's output, the example's class, and the target vector.
- Write down the formulas defining the *error rate* and the *mean square error*.

[2]More precisely, the outputs will only *approach* 1 and 0 because the sigmoid function is bounded by the *open* interval, $(0, 1)$.

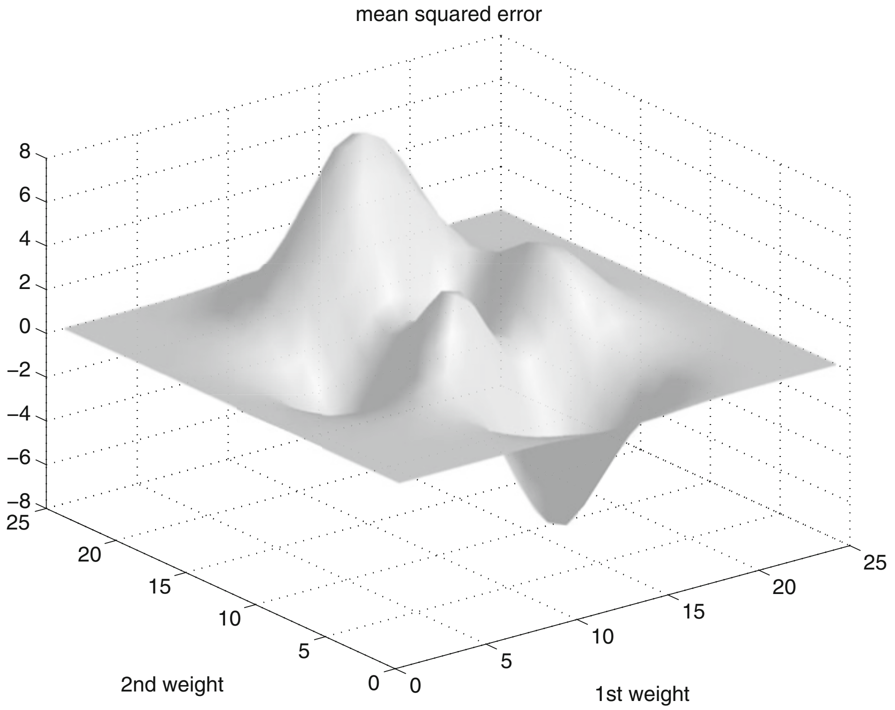


Fig. 5.3 For a given example, each set of weights implies a certain mean square error. Training should reduce this error as quickly as possible

5.3 Backpropagation of Error

In the *multilayer perceptron*, the parameters to affect the network's behavior are the sets of weights, $w_{ji}^{(1)}$ and $w_{kj}^{(2)}$. The task for machine learning is to find for these weights such values that will optimize the network's classification performance. Just like in the case of linear classifiers, this is achieved by *training*. This section is devoted to one popular technique capable of accomplishing this task.

The general scenario. In principle, the procedure is the same as in the previous chapter. At the beginning, the weights are *initialized* to small random numbers, typically from the interval $(-0.1, 0.1)$. After this, the training examples are presented, one by one, and each of them is forward-propagated to the network's output. The discrepancy between this output and the example's target vector then tells us how to modify the weights (see below). After the weight modification, the next example is presented. When the last training example has been reached, one *epoch* has been completed. In *multilayer perceptrons*, the number of epochs needed for successful training is much greater than in the case of linear classifiers: it can be thousands, tens of thousands, even more.

The gradient descent. Before we proceed to the concrete formulas for weight adjustment, we need to develop a better understanding of the problem's nature. Figure 5.3 will help us. The vertical axis represents the mean square error, expressed as a function of the network's weights (plotted along the horizontal axes). For graphical convenience, we assume that there are only two weights. This, of course, is unrealistic to say the least. But if we want an instructive example, we simply cannot afford more dimensions—we can hardly visualize ten dimensions, can we? The message we want to drive home at this point is that the error function can be imagined as a kind of a "landscape" whose "valleys" represent the function's *local minima*. The deepest of them is the *global minimum*, and this is what the training procedure is, in the ideal case, expected to reach; more specifically, it should find the set of weights corresponding to the global minimum.

A quick glance at Fig. 5.3 tells us that any pair of weights defines for the given training example a concrete location on the landscape, typically somewhere on one of the slopes. Any weight change will then result in different coordinates along the horizontal axes, and thus a different location on the error function. Where exactly this new location is, whether "up" or "down" the slope, will depend on how much, and in what direction, each of the weights has changed. For instance, it may be that increasing both w_1 and w_2 by 0.1 will lead only to a minor reduction of the mean square error; whereas increasing w_1 by 0.3 and w_2 by 0.1 will reduce it considerably.

In the technique discussed here, we want weight-changes that will bring about the *steepest descent* along the error function. Recalling the terminology from Chap. 1, this is a job for *hill-climbing* search. The best-known technique used to this end in *multilayer perceptrons* is *backpropagation of error*.

Backpropation of error. The specific weight-adjusting formulas can be derived from Eq. 5.2 by finding the function's gradient. However, as this book is meant for practitioners, and not for mathematicians, we will skip the derivation, and focus instead on explaining the learning procedure's behavior.

To begin with, it is reasonable to assume that the individual neurons differ in their contributions to the overall error. Some of them "spoil the game" more than the others. If this is the case, the reader will agree that the links leading to these neurons should undergo greater weight changes than the links leading to less offending neurons.

Fortunately, each neuron's amount of "responsibility" for the overall error can be easily obtained. Generally speaking, the concrete choice of formulas depends on what transfer function has been used. In the case of the `sigmoid` (see Eq. 5.1), the responsibility is calculated as follows:

Output-layer neurons: $\delta_i^{(1)} = y_i(1 - y_i)(t_i - y_i)$
Here, $(t_i - y_i)$ is the difference between the i-th output and the corresponding target value. This difference is multiplied by $y_i(1 - y_i)$, a term whose minimum value is reached when $y_1 = 0$ or $y_i = 1$ (a "strong opinion" as to whether $\mathbf{x}$ should or should not be labeled with the i-th class); the term is maximized when $y_i = 0.5$, in which case the "opinion" can be deemed neutral. Note that the sign of $\delta_i^{(1)}$ depends only on $(t_i - y_i)$ because $y_i(1 - y_i)$ is always positive.

Table 5.2 *Backpropagation of error* in a neural network with one hidden layer

```
1. Present example x to the input layer and propagate it through the network.
2. Let $\mathbf{y} = (y_1, \ldots y_m)$ be the output vector, and let $\mathbf{t}(\mathbf{x}) = (t_1, \ldots t_m)$ be the target vector.
3. For each output neuron, calculate its responsibility, $\delta_i^{(1)}$, for the network's error:
   $\delta_i^{(1)} = y_i(1 - y_i)(t_i - y_i)$
4. For each hidden neuron, calculate its responsibility, $\delta_j^{(2)}$, for the network's error. While doing
   so, use the responsibilities, $\delta_i^{(1)}$, of the output neurons as obtained in the previous step.
   $\delta_j^{(2)} = h_j(1 - h_j) \sum_i \delta_i^{(1)} w_{ji}$
5. Update the weights using the following formulas, where $\eta$ is the learning rate:
   output layer: $w_{ji}^{(1)} := w_{ji}^{(1)} + \eta\delta_i^{(1)} h_j$;   $h_j$: the output of the j-th hidden neuron
   hidden layer: $w_{kj}^{(2)} := w_{kj}^{(2)} + \eta\delta_j^{(2)} x_k$;   $x_k$: the value of the k-th attribute
6. Unless a termination criterion has been satisfied, return to step 1.
```

Hidden-layer neurons: $\delta_j^{(2)} = h_j(1 - h_j) \sum_i \delta_i^{(1)} w_{ji}$
The responsibilities of the hidden neurons are calculated by "backpropagating" the output-neurons' responsibilities obtained in the previous step. This is the role of the term $\sum_i \delta_i^{(1)} w_{ji}$. Note that each $\delta_i^{(1)}$ (the responsibility of the i-th output neuron) is multiplied by the weight of the link connecting the i-th output neurons to the j-th hidden neuron. The weighted sum is multiplied by, $h_j(1 - h_j)$, essentially the same term as the one used in the previous step, except that h_j has taken the place of y_i.

Weight updates. Now that we know the responsibilities of the individual neurons, we are ready to update the weights of the links leading to them. Similarly to *perceptron learning*, an additive rule is used:

$$\text{output-layer neurons: } w_{ji}^{(1)} := w_{ji}^{(1)} + \eta\delta_i^{(1)} h_j$$
$$\text{hidden-layer neurons: } w_{kj}^{(2)} := w_{kj}^{(2)} + \eta\delta_j^{(2)} x_k$$

The extent of weight correction is therefore determined by $\eta\delta_i^{(1)} h_j$ or $\eta\delta_j^{(2)} x_k$. Two observations can be made. First, the neurons' responsibilities, $\delta_i^{(1)}$ or $\delta_j^{(2)}$, are multiplied by η, the *learning rate* which, theoretically speaking, should be selected from the unit interval, $\eta \in (0, 1)$; however, practical implementations usually rely on smaller values, typically less then 0.1. Second, the terms are also multiplied by $h_j \in (0, 1)$ and $x_k \in [0, 1]$, respectively. The correction is therefore quite small, but its effect is relative. If the added term's value is, say, 0.02, then smaller weights, such as $w_{ij}^{(1)} = 0.01$, will be affected more significantly than greater weights such as $w_{ij}^{(1)} = 1.8$.

The whole training procedure is summarized by the pseudocode in Table 5.2. The reader will benefit also from taking a closer look at the example given in Table 5.3 that provides all the necessary details of how the weights are updated in response to one training example.

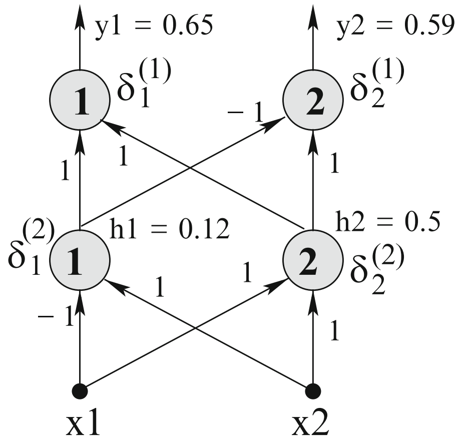
What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Why is the training technique called "backpropagation of error"? What is the rationale behind the need to establish the "neurons' responsibilities"?
- Discuss the behaviors of the formulas for the calculation of the responsibilities of the neurons in the individual layers.
- Explain the behaviors of the formulas used to update the weights. Mention some critical aspects of these formulas.

Table 5.3 Illustration of the backropagation of error

Task. In the neural network below, let the transfer function be $f(\Sigma) = \frac{1}{1+e^{-\Sigma}}$. Using backpropagation of error (with $\eta = 0.1$), show how the weights are modified after the presentation of the following example: $[\mathbf{x}, \mathbf{t}(\mathbf{x})] = [(1, -1), (1, 0)]$



Forward propagation.

The picture shows the state after forward propagation when the signals leaving the hidden and the output neurons have been calculated as follows:

- $h_1 = \frac{1}{1+e^{-(-2)}} = 0.12$
  $h_2 = \frac{1}{1+e^{0}} = 0.5$
  $y_1 = \frac{1}{1+e^{-(0.12+0.5)}} = 0.65$
  $y_2 = \frac{1}{1+e^{-(-0.12+0.5)}} = 0.59$

(the solution continues on the next page)

Backpropagation of error (cont. from the previous page)

The target vector being $\mathbf{t}(\mathbf{x}) = (1, 0)$, and the output vector $\mathbf{y} = (0.65, 0.59)$, the task is to establish each neuron's responsibility for the output error. Here are the calculations for the output neurons:

- $\delta_1^{(1)} = y_1(1-y_1)(t_1-y_1) = 0.65(1-0.65)(1-0.65) = 0.0796$
  $\delta_2^{(1)} = y_2(1-y_2)(t_2-y_2) = 0.59(1-0.59)(0-0.59) = -0.1427$

Using these values, we calculate the responsibilities of the hidden neurons. Note that we will first calculate (and denote by Δ_1 and Δ_2) the weighted sums, $\sum_i \delta_i^{(1)} w_{ij}^{(1)}$, for each of the two hidden neurons.

- $\Delta_1 = \delta_1^{(1)} w_{11}^{(1)} + \delta_2^{(1)} w_{12}^{(1)} = 0.0796 \times 1 + (-0.1427) \times (-1) = 0.2223$
  $\Delta_2 = \delta_1^{(1)} w_{21}^{(1)} + \delta_2^{(1)} w_{22}^{(1)} = 0.0796 \times 1 + (-0.1427) \times 1 = -0.0631$
  $\delta_1^{(2)} = h_1(1-h_1)\Delta_1 = 0.12(1-0.12) \times 0.2223 = -0.0235$
  $\delta_2^{(2)} = h_2(1-h_2)\Delta_2 = 0.5(1-0.5) \times (-0.0631) = 0.0158$

Once the responsibilities are known, the weight modifications are straightforward:

- $w_{11}^{(1)} = w_{11}^{(1)} + \eta\delta_1^{(1)} h_1 = 1 + 0.1 \times 0.0796 \times 0.12 = 1.00096$
  $w_{21}^{(1)} = w_{21}^{(1)} + \eta\delta_1^{(1)} h_2 = 1 + 0.1 \times 0.0796 \times 0.5 = 1.00398$
  $w_{12}^{(1)} = w_{12}^{(1)} + \eta\delta_2^{(1)} h_1 = -1 + 0.1 \times (-0.1427) \times 0.12 = -1.0017$
  $w_{22}^{(1)} = w_{22}^{(1)} + \eta\delta_2^{(1)} h_2 = 1 + 0.1 \times (-0.1427) \times 0.5 = 0.9929$
- $w_{11}^{(2)} = w_{11}^{(2)} + \eta\delta_1^{(2)} x_1 = -1 + 0.1 \times (-0.0235) \times 1 = -1.0024$
  $w_{21}^{(2)} = w_{21}^{(2)} + \eta\delta_1^{(2)} x_2 = 1 + 0.1 \times (-0.0235) \times (-1) = 1.0024$
  $w_{12}^{(2)} = w_{12}^{(2)} + \eta\delta_2^{(2)} x_1 = 1 + 0.1 \times 0.0158 \times 1 = 1.0016$
  $w_{22}^{(2)} = w_{22}^{(2)} + \eta\delta_2^{(2)} x_2 = 1 + 0.1 \times 0.0158 \times (-1) = 0.9984$

The weights having been updated, the network is ready for the next example.

5.4 Special Aspects of Multilayer Perceptrons

Limited space prevents us from the detailed investigation of the many features that make the training of *multilayer perceptrons* more art than science. To do justice to all of them, another chapter at least the size of this one would be needed. Still, the knowledge of certain critical aspects is vital if the training is ever to succeed. Let us therefore briefly survey some of the more important ones.

Computational costs. Backpropagation of error is computationally expensive. Upon the presentation of an example, the responsibility of each individual neuron has to be calculated, and the weights then modified accordingly. This has to be repeated for all training examples, usually for many epochs. To get an idea of the real costs of all this, consider a network that is to classify examples described by 100 attributes, a fairly realistic case. If there are 100 hidden neurons, then the number of weights in this lower layer is $100 \times 100 = 10^4$. This then is the number of weight-changes after each training example. Note that the upper-layer weights can be neglected, in these calculations, as long as the number of classes is small. For instance, in a domain with three classes, the number of upper-layer weights is $100 \times 3 = 300$ which is much less than 10^4.

Suppose the training set consists of 10^5 examples, and suppose that the training will continue for 10^4 epochs. In this event, the number of weight-updates will be $10^4 \times 10^5 \times 10^4 = 10^{13}$. This looks like a whole lot, but many applications are even more demanding. Some fairly ingenious methods to make the training more efficient have therefore been developed. These, however, are outside the scope of our interest here.

Target values revisited. For simplicity, we have so far assumed that each target value is either 1 or 0. This may not be the best choice. For one thing, these values can never be reached by a neuron's output, y_i. Moreover, the weight changes in the vicinity of these two extremes are miniscule because the calculation of the output-neuron's responsibility, $\delta_i^{(1)} = y_i(1 - y_i)(t_i - y_i)$, returns a value very close zero whenever y_i approaches 0 or 1. Finally, we know that the classifier chooses the class whose output neuron has returned the highest value. The individual neuron's output precision therefore does not matter much; more important is the comparison with the other outputs. If the forward propagation results in $\mathbf{y} = (0.9, 0.1, 0.2)$, then the example is bound to be labeled with the first class (the one supported by $y_i = 0.9$), and this decision will not be swayed by minor weight changes.

In view of these arguments, more appropriate values for the target are recommended: for instance, $t_i(\mathbf{x}) = 0.8$ if the example belongs to the i-th class, and $t_i(\mathbf{x}) = 0.2$ if it does not. Suppose there are three classes, C_1, C_2, and C_3, and suppose $c(\mathbf{x}) = C_1$. In this case, the target vector will be defined as $\mathbf{t}(\mathbf{x}) = (0.8, 0.2, 0.2)$. Both 0.8 and 0.2 find themselves in regions of relatively high sensitivity of the sigmoid function, and as such will eliminate most of the concerns raised in the previous paragraph.

Local minima. Figure 5.3 illustrated the main drawback of the gradient-descent approach when adopted by *multilayer perceptron* training. The weights are changed in a way that guarantees descent along the steepest slope. But once the bottom of a *local minimum* has been reached, there is nowhere else to go—which is awkward: after all, the ultimate goal is to reach the *global minimum*. Two things are needed here: first, a mechanism to recognize the local minimum; second, a method to recover from having fallen into one.

One possibility of timely identification of local minima during training is to keep track of the mean square error, and to sum it up over the entire training set at the end of each epoch. Under normal circumstances, this sum tends to go down from one epoch to another. Once it seems to have reached a plateau where hardly any error reduction can be observed, the learning process is suspected of being trapped in a local minimum.

Techniques to overcome this difficulty usually rely on adaptive learning rates (see below), and on adding new hidden neurons (see Sect. 5.5). Generally speaking, the problem is less critical in networks with many hidden neurons. Also, local minima tend to be shallower, and less frequent, if all weights are very small, say, from the interval $(-0.01, 0.01)$.

Adaptive learning rate. While describing backpropagation of error, we assumed that the user-set learning rate, η, was a constant. This, however, is rarely the case in realistic applications. Very often, the training starts with a large η which then gradually decreases in time. The motivation is easy to guess. At the beginning, greater weight-changes reduce the number of epochs, and they may even help the learner to "jump over" some local minima. Later on, however, this large η might lead to "overshooting" the global minimum, and this is why its value should be decreased. If we express the learning rate as a function of time, $\eta(t)$, where t tells us which epoch the training is now going through, then the following negative-exponential formula will gradually reduce the learning rate (α is the slope of the negative exponential, and $\eta(0)$ is the learning rate's initial value):

$$\eta(t) = \eta(0)e^{-\alpha t} \tag{5.4}$$

It should perhaps be noted that some advanced weight-changing formulas are capable of reflecting "current tendencies." For instance, it is quite popular to implement "momentum": if the last two weight changes were both positive (or both negative), it makes sense to increase the weight-changing step; if, conversely, a positive change was followed by a negative change (of vice versa), the step should be reduced so as to prevent overshooting.

Overtraining. *Multilayer perceptrons* share with polynomial classifiers one unpleasant property. Theoretically speaking, they are capable of modeling *any* decision surface, and this makes them prone to overfitting the training data. The reader remembers that overfitting typically means perfect classification of noisy training examples, which is inevitably followed by disappointing performance in the future.

For small *multilayer perceptrons*, this problem is not as painful; they are not flexible enough to overfit. But as the number of hidden neurons increases, the network gains in flexibility, and overfitting can become a real concern. However, as we will learn in the next section, this does *not* mean that we should always prefer small networks!

There is a simple way to discover whether the training process has reached the "overfitting" stage. If the training set is big enough, we can afford to leave aside some 10–20 % examples. These will never be used for backpropation of error; rather, after each epoch, the sum of mean square errors on these withheld examples is calculated. At the beginning, the sum will tend to go down, but only up to a certain moment; then it starts growing again, alerting the engineer that the training has begun to overfit the data.

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What do you know about the computational costs of the technique of backpropagation of error?
- Explain why this section recommended the values of the target vector to be chosen from $\{0.8, 0.2\}$ instead of from $\{1,0\}$.
- Discuss the problem of local minima. Why do they represent such a problem for training? How can we reduce the danger of getting trapped in one?
- What are the benefits of an adaptive learning rate? What formula has been recommended for it?
- What do you know about the danger that the training of a neural network might result in overfitting the training data?

5.5 Architectural Issues

So far, one important question has been neglected: how many hidden neurons to use? If there are only one or two, the network will lack flexibility; not only will it be unable to model a complicated decision surface; the training of this network will be prone to get stuck in a local minimum. At the other extreme, using thousands of neurons will not only increase computational costs because of the need to train so many neurons. The network will be more flexible than needed. As a result, it will easily overfit the data. As usual, in situations of this kind, some compromise has to be found.

Performance versus size. Suppose you decide to run the following experiment. The available set of preclassified examples is divided into two parts, one for training, the other for testing. Training is applied to several neural networks, each with a different number of hidden neurons. The networks are trained until no reduction of the training-set error rate is observed. After this, the error rate on the testing data is measured.

Optimum number of neurons. When plotted in a graph, the results will typically look something like the situation depicted in Fig. 5.4. Here, the horizontal axis represents the number of hidden neurons; the vertical axis, the error rate measured on the testing set. Typically, the error rate will be high in the case of very small networks because these lack adequate flexibility, and also suffer from the danger of getting stuck in local minima. These two weaknesses can be mitigated if we increase the number of hidden neurons. As shown in the graph, the larger networks then exhibit lower error rates. But then, networks that are too large are vulnerable to overfitting. This is why, after a certain point, the testing-set error starts growing again (the right tail of the graph).

The precise shape of the curve depends on the complexity of the training data. Sometimes, the error rate is minimized when the network contains no more than 3–5 hidden neurons. In other domains, however, the minimum is reached only when hundreds of hidden neurons are used. Yet another case worth mentioning is the situation where the training examples are completely noise-free. In a domain of this kind, overfitting may never become an issue, and the curve's right tail may not grow at all.

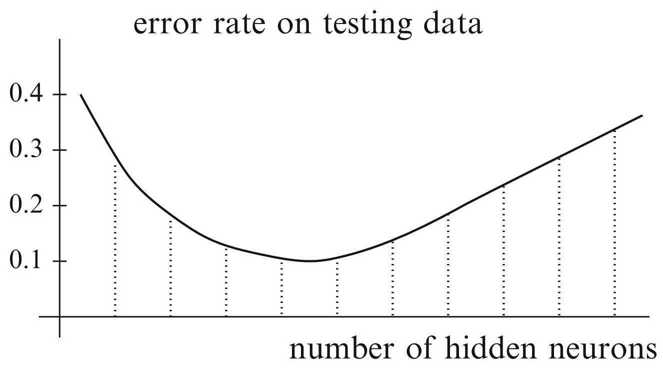


Fig. 5.4 The error rate measured on testing examples depends on the number of neurons in the hidden layer

Table 5.4 Gradual search for a good size of the hidden layer

```
1. At the beginning, use only a few hidden neurons, say, five.
2. Train the network until the mean square error no longer seems to improve.
3. At this moment, add a few (say, three) neurons to the hidden layer, each with randomly
   initialized weights, and resume training.
4. Repeat the previous two steps until a termination criterion has been satisfied. For instance,
   this can be the case when the new addition does not result in a significant error reduction, or
   when the hidden layer exceeds a user-set maximum size.
```

Search for appropriate size. The scenario described above is too expensive to be employed in practical applications. After all, we have no idea whether we will need just a few neurons, or dozens of them, or hundreds, and we may have to re-run the computationally intensive training algorithm a great many times before being able to establish the ideal size. Instead, we would like to have at our disposal a technique capable of finding the appropriate size more efficiently.

One such technique is summarized by the pseudocode in Table 5.4. The idea is to start with a very small network that only has a few hidden neurons. After each epoch, the learning algorithm checks the sum of the mean square errors observed on the training set. This sum of errors is likely to keep decreasing with the growing number of epochs—but only up to a certain point. When this is reached, the network's performance no longer improves, either because of its insufficient flexibility that makes correct classification impossible, or because it "fell" into a local minimum. When this is observed, a few more neurons with randomly initialized weights are added, and the training is resumed.

Usually, the added flexibility makes further error-reduction possible. In the illustration from Fig. 5.5, the error rate "stalled" on two occasions; adding new hidden neurons then provided the necessary new flexibility.

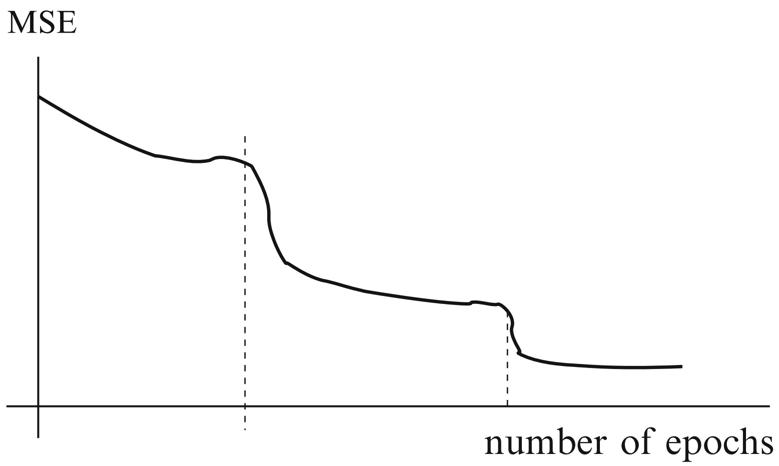


Fig. 5.5 When the training-set mean square error (MSE) does not seem to go down, further improvement can be achieved by adding new hidden neurons. In this particular case, this has been done twice

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Discuss the shape of the curve from Fig. 5.4. What are the shortcomings of very small and very large networks?
- Explain the algorithm that searches for a reasonable size of the hidden layer. What main difficulties does it face?

5.6 Radial Basis Function Networks

The behavior of an output neuron in *multilayer perceptrons* is similar to that of a linear classifier. As such, it may be expected to fare poorly in domains with classes that are not linearly separable. In the context of the neural networks, however, this limitation is not necessarily hurtful. Thing is, the original examples have been transformed by the sigmoid functions in the hidden layer. Consequently, the neurons in the output layer deal with new "attributes," those obtained by this transformation. In the process of training, these transformed examples (the outputs of the hidden neurons) may become linearly separable so that the output-layer neurons can separate the two classes without difficulties.

The alternative. There is another way of transforming the attribute values; by using the so-called $\underline{r}$adial-$\underline{b}$asis $\underline{f}$unction, RBR, as the transfer function employed

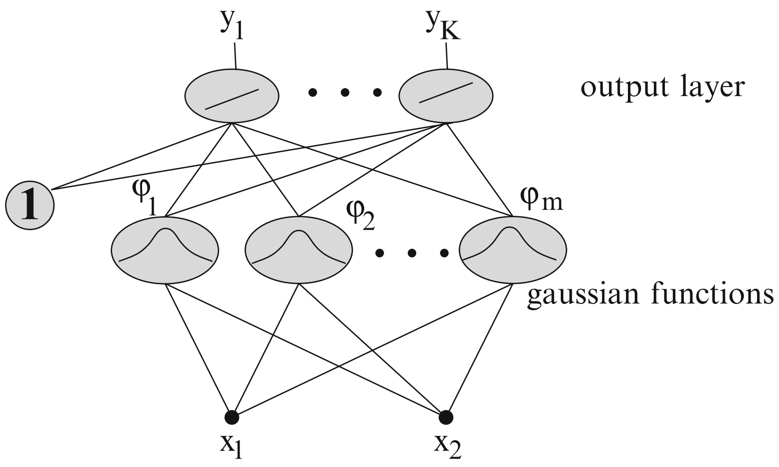


Fig. 5.6 Radial-basis function network

by the hidden-layer neurons. This is the case of the network depicted in Fig. 5.6. An example presented to the input is passed through a set of neurons that each return a value denoted as φ_j.

Radial-basis function, RBF. In essence, this is based on the normal distribution that we already know from Chap. 2. Suppose that the attributes describing the examples all fall into some reasonably-sized interval, say $[-1, 1]$. For a given variance, σ^2, the following equation defines the n-dimensional gaussian surface centered at $\boldsymbol{\mu_j} = [\mu_{j1}, \ldots \mu_{jn}]$:

$$\varphi_j(\mathbf{x}) = \exp\{-\frac{\Sigma_{i=1}^{n}(x_i - \mu_{ji})^2}{2\sigma^2}\} \tag{5.5}$$

In a sense, $\varphi_j(\mathbf{x})$ measures similarity between the example vector, $\mathbf{x}$, and the gaussian center, $\boldsymbol{\mu_j}$: the larger the distance between the two, the smaller the value of $\varphi_j(\mathbf{x})$. If $\mathbf{x}$ is to be classified, the network first redescribes it as $\boldsymbol{\varphi}(\mathbf{x}) = [\varphi_1(\mathbf{x}), \ldots, \varphi_m(\mathbf{x})]$. The output signal of the i-th output neuron is then calculated as $y_i = \sum_{j=0}^{m} w_{ji}\varphi_j(\mathbf{x})$, where w_{ji} is the weight of the link from the j-th hidden neuron to the i-th output neuron (the weights w_{0i} are connected to a fixed $\varphi_0 = 1$). This output signal being interpreted as the amount of evidence supporting the i-th class, the example is labeled with the i-th class if $y_i = \max_k(y_k)$.

Output-layer weights. It is relatively simple to establish the output-layer *weights*, w_{ij}. Since there is only one layer of weights to be trained, we can just as well rely on the *perceptron learning* algorithm described in Chap. 4, applying it to examples whose descriptions have been transformed by the RBF functions in the hidden-layer neurons.

Gaussian centers. It is a common practice to identify the gaussian centers, $\boldsymbol{\mu}_j$, with the individual training examples. If the training set is small, we can simply use one hidden neuron per training example. In many realistic applications, though, the training sets are much bigger, which can mean thousands of hidden neurons, or even more. Realizing that such large networks are unwieldy, many engineers prefer to select for the centers only a small subset of the training examples. Very often, a random choice is enough. Another possibility to identify groups of "similar" vectors, and then use for each RBF neuron the center of one group.

RBF-based support vector machines. The RBF neurons transform the original example description into a new vector that consists of the values, $\phi_1, \ldots \phi_m$. Most of the time, this transformation increases the chances that the examples thus transformed will be linearly separable. This makes it possible to subject them to a linear classifier whose weights are trained by *perceptron learning*.

This is perhaps an appropriate place to mention that it is also quite popular to apply to the transformed examples the idea of the *support vector machine* introduced in Sect. 4.7. In this event, the resulting machine-learning tool is usually referred to as RBF-based SVM. Especially in domains where the boundary between the classes is highly non-linear, this classifier is more powerful than the plain (linear) SVM.

Computational costs. RBF-network training consists of two steps. First, the centers of the radial-basis functions (the gaussians) are selected. Second, the output-neurons' weights are obtained by training. Since there is only one layer to train, the process is computationally much less intensive than the training in *multilayer perceptrons*.

What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the principle of the radial-basis function network. In what aspects does it differ from the *multilayer perceptron*?
- How many weights need to be trained in a radial-basis function network? How can the training be accomplished?
- What are the possibilities for the construction of the gaussian hidden layer?

5.7 Summary and Historical Remarks

- The basic unit of a *multilayer perceptron* is a *neuron*. The neuron accepts a weighted sum of inputs, and subjects this sum to a *transfer function*. Several different transfer functions can be used. The one chosen in this chapter is the *sigmoid* defined by the following equation where Σ is the weighted sum of inputs:

$$f(\Sigma) = \frac{1}{1 + e^{-\Sigma}}$$

Usually, all neurons use the same transfer function.

- The simple version of the *multilayer perceptron* described in this chapter consists of one output layer and one hidden layer of neurons. Neurons in adjacent layers are fully interconnected; but there are no connections between neurons in the same layer. An example presented to the network's input is *forward-propagated* to its output, implementing, in principle, the following function:

$$y_i = f(\sum_j w_{ji}^{(1)} f(\sum_k w_{kj}^{(2)} x_k))$$

  Here, $w_{ji}^{(1)}$ and $w_{kj}^{(2)}$ are the weights of the output and the hidden neurons, respectively, and f is the sigmoid function.
- The training of *multilayer perceptron* is accomplished by a technique known as *backpropagation of error*. For each training example, the technique first establishes each neuron's individual responsibility for the network's overall error, and then updates the weights according to these responsibilities.

  Here is how the responsibilities are calculated:

  output neurons: $\delta_i^{(1)} = y_i(1 - y_i)(t_i - y_i)$
  hidden neurons: $\delta_j^{(2)} = h_j(1 - h_j) \sum_i \delta_i^{(1)} w_{ij}$

  Here is how the weights are updated:

  output layer: $w_{ji}^{(1)} := w_{ji}^{(1)} + \eta \delta_i^{(1)} h_j$
  hidden layer: $w_{kj}^{(2)} := w_{kj}^{(2)} + \eta \delta_j^{(2)} x_k$

- Certain aspects of *backpropagation by error* have been discussed here. Among these, the most important are computational costs, the existence of local minima, adaptive learning rate, the danger of overfitting, and the problems of how to determine the size of the hidden layer.
- An alternative is the *radial-basis function* (RBF) network. For the transfer function at the hidden-layer neurons, the gaussian function is used. The output-layer neurons often use the step function (in principle, the linear classifier), or simply a linear function of the inputs.
- In RBF networks, each gaussian center is identified with one training example. If there are too many such examples, a random choice can be made. The gaussian's standard deviation is in this simple version set to $\sigma^2 = 1$,
- The output-layer neurons in RBF networks can be trained by perceptrons learning. Only one layer of weights needs to be trained.
- Sometimes, the idea of *support vector machine (SVM)* is applied to the outputs of hidden neurons. The resulting tool is then called the RBF-based SVM.

Historical Remarks. Research of neural networks was famously delayed by the sceptical views formulated by Minsky and Papert [56]. The pessimism expressed by such famous authors was probably the main reason why an early version of neural-network training by Bryson and Ho [11] was largely overlooked, a fate soon to be shared by an independent successful attempt by Werbos [82]. It was only after the publication of the groundbreaking volumes by Rumelhart and McClelland [70], where the algorithm was independently re-invented, that the field of artificial neural networks became established as a respectable scientific discipline. The gradual growth of the multilayer perceptron was proposed by Ash [1]. The idea of radial-basis functions was first cast in the neural-network setting by Broomhead and Lowe [10].

5.8 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

Exercises

1. Return to the illustration of backpropagation of error in Table 5.3. Using only a pen, paper, and calculator, repeat the calculations for a slightly different training example: $\mathbf{x} = (-1, -1), t(\mathbf{x}) = (0, 1)$.
2. Hand-simulating backpropation of error as in the previous example, repeat the calculation for the following two cases:

   High output-layer weights: $w_{11}^{(1)} = 3.0, w_{12}^{(1)} = -3.0, w_{21}^{(1)} = 3.0, w_{22}^{(1)} = 3.0$
   Small output-layer weights: $w_{11}^{(1)} = 0.3, w_{12}^{(1)} = -0.3, w_{21}^{(1)} = 0.3, w_{22}^{(1)} = 0.3$

   Observe the relative changes in the weights in each case.
3. Consider a training set containing of 10^5 examples described by 1,000 attributes. What will be the computational costs of training a *multilayer perceptron* with 1,000 hidden neurons and 10 output neurons for 10^5 epochs?

Give it Some Thought

1. How will you generalize the technique of backpropagation of error so that it can be used in a *multilayer perceptron* with more that one hidden layer?

2. Section 5.1 suggested that all attributes should be normalized here to the interval $[-1.0, 1.0]$. How will the network's classification and training be affected if the attributes are not so normalized? (hint: this has something to do with the sigmoid function)
3. Discuss the similarities and differences of the classification procedures used in radial-basis functions and in *multilayer perceptrons*.
4. Compare the advantages and disadvantages of radial-basis function networks in comparison with *multilayer perceptrons*.

Computer Assignments

1. Write a program that implements backpropagation of error for a predefined number of output and hidden neurons. Use a fixed learning rate, η.
2. Apply the program implemented in the previous task to some benchmark domains from the UCI repository.[3] Experiment with different values of η, and see how they affect the speed of convergence.
3. For a given data set, experiment with different numbers of hidden neurons in the *multilayer perceptron*, and observe how they affect the network's ability to learn.
4. Again, experiment with different numbers of hidden neurons. This time, focus on computational costs. How many epochs are needed before the network converges? Look also at the evolution of the error rate.
5. Write a program that will, for a given training set, create a radial-basis function network. For large training sets, select randomly the examples that will define the gaussian centers.
6. Apply the program implemented in the previous task to some benchmark domains from the UCI repository.

[3] www.ics.uci.edu/~mlearn/MLRepository.html