Feed-forward mapping networks

In this chapter we explore the ability of feedforward neural networks to represent mapping functions. Mapping functions are important in many brain processes and have dominated models in cognitive science in the form of multilayer perceptrons. We start exploring the effects of choosing appropriate values for synaptic weights through learning algorithms. While feedforward networks are not enough to explain cognitive functions alone, they are an important ingredient of brain-style information processing and have contributed greatly to the development of statistical learning theory. This chapter includes some review about concepts of machine learning and more recent developments such as support vector machines. These less biological discussions are included for their contribution to the general understanding of learning theories, and also since such tools have become important methods for analysing data in neuroscience.

6.1 The simple perceptron

6.1.1 Optical character recognition (OCR)

To illustrate the abilities of networks introduced in this chapter, we follow an example of optical character recognition (OCR). OCR is the process of (optically) scanning an image and interpreting this digital image so that the computer ‘understands’ its meaning. For example, if we scan an image of the letter A, than we would like to store a binary number 1000001 in the computer. This binary number is a standard representation for encoding the letter A in the American Standard Code for Information Interchange (ASCII). Once the letter is represented with this code, a program can use this code to infer meaning and use this to perform specific tasks, such as displaying a representation of the letter on screen, printing the letter with different fonts on paper, or to look up words made out of such letters in a dictionary. The transformation of the digital image representation to the representation of the meaning of the image on a computer is a very challenging engineering task.

While there are now some OCR programs on the market, humans typically outperform these systems easily. Also, character recognition is a relatively easy task for humans, and it is thus interesting to compare OCR to functions in the human perceptual system. While we will stress later that the human system relies heavily on top-down expectation (Section 10.3), the feedforward systems in this chapter are an important component of information processing in the brain. To discuss the example in more detail, it is useful to distinguish two
processes in the perception of a letter, the physical sensing of an image of a letter, and attaching meaning to such an image. This is similar to the OCR example above, which includes a scanning phase, which converts the printed image into a binary representation, and an recognition phase, which relies on the transformation of a binary image into a meaningful representation. When we read a letter, each letter has to be transformed from the representation on paper to a representation in the brain which can be interpreted by the brain.

Let's discuss the example of recognizing the letter 'A' as shown on the left side of Fig. 6.1. The first step towards perception of this letter is to get the signal into the brain. This is achieved by an optical sensor called eye with photoceptors able to transduce light falling on to the retina into signals that are transmitted to the brain. We approximate this process with a simplified digitizing model retina of only $10 \times 10 = 100$ photoceptors. Each of these photoceptors transmits a signal with value 1 if the area covered by the receptor (its receptive field) is covered at least partially by the image. In this way we end up with the digitized version of the image shown on the right side of Fig. 6.1.

This model is certainly a crude approximation of a human eye. The resolution in the human retina is much higher; a typical retina has around 125 million photoceptors. Also, the receptors in a human retina are not homogeneously distributed, as in our example, but have highest density in the centre of the visual field called the fovea. In addition, it is well known that much more sophisticated signal processing goes on in a human eye through several neuronal layers with specialized neurons before a signal leaves the eye through the axons of ganglion cells. However, this crude model is simply intended to illustrate a general scheme, and there is no need to complicate the model with more realistic but irrelevant details for the discussions in this section.

The output of the simple model retina is a collection of values that can be collected in a vector rather than the array shown in Fig. 6.1. To do this, we give each model neuron an individual number and write the value of this node into a large column at a position corresponding to this number of the node (see Fig. 6.2). We call this vector the sensory feature vector, $\mathbf{x}$. Individual components are written with an index such as $x_i$, where the index $i$ numbers the different feature values. Feature values can be binary, as in our example, real-valued, or any other symbolic representation. The number of feature values defines the dimensionality of the feature space, which is often very large (100 in our example; over a million for the visual feature vector generated by one eye if we use as basis the number ganglion cells in humans). It is important to keep in mind that the precise form of sensory feature vectors depends on the specific functionality of the feature sensors (the retina in our example) and the
related encoding procedures. If the form of the feature vector is crucial, we have either to model the biological details of the sensory system or to make valid assumptions that reflect the biological system. The precise form of the sensory feature vector is not important to demonstrate the principal ideas presented in this chapter.

6.1.2 Mapping functions

A sensory feature vector is the necessary input to any object recognition system. Given such a sensory feature vector we can formulate the character recognition process as the problem of mapping this sensory feature vector onto a set of symbols representing the internal representation of the object. This internal representation corresponds, for example, to the ASCII code of the letter in the computer example, or to a largely unknown representation in the language area in our brain. In our example above we used the bit vector of length 7 (ASCII) for the internal representation of A, and we could use the corresponding ASCII code for other letters. Or we could use the number from 1 to 26, which corresponds to the position of the letter in the alphabet.

At this point we have reduced the recognition process to a vector function, or mapping for short. The OCR example is a mapping from a binary feature vector to binary vector for the computer representation of the meaning. Generally, we can define a mapping as a vector function \( f \) from a vector \( x \) to another vector \( y \) as

\[
f : x \in \mathbb{S}_1^n \rightarrow y \in \mathbb{S}_2^m,
\]

(6.1)

where \( n \) is the dimensionality of the sensory feature space (\( n = 100 \) in our example), and \( m \) is the dimensionality of the internal object representation space (\( m = 1 \) in our example). \( \mathbb{S}_1 \) and \( \mathbb{S}_2 \) are the sets of possible values for each individual component of the vectors. For example, the components of the vector can consist of binary values (\( \mathbb{S} = \{0, 1\} \)), discrete values (\( \mathbb{S} = \mathbb{N} \)), or real values (\( \mathbb{S} = \mathbb{R} \)) such as the firing rates of neurons. In the example above we mentioned a subset of the natural numbers, \( \mathbb{S}_2 = [1, \ldots, 26] \), to represent the meaning of the 26 letters in the alphabet.
How can we realize a mapping function? There are several possibilities, which we will discuss in turn. Our aim is to understand how the brain achieves this mapping, but it is instructive to think about some of the possibilities. The simplest solution is to construct a look-up table. A look-up table is a large table that lists for all possible sensory input vectors the corresponding internal representations. To illustrate this we further reduce the size of the feature vector and consider a feature vector that has only two components. Hence, the dimensionality of the sensory feature space is only 2 (not 100 as before). Let’s call the first feature value \( x_1 \) and the second feature value \( x_2 \). Each sensory feature value can only have two possible values, 0 and 1 (binary feature vector). Thus, there are four possible combinations of feature values, \{ (0 0), (0 1), (1 0), (1 1) \}. A look-up table for a function \( y(x) \) lists the desired output value \( y \) for each combination of the input (feature) values. An example is given in Table 6.1 representing one possible binary mapping function of two inputs, the Boolean AND function.

With binary feature values we can have \( n_c = 2^2 = 4 \) different combinations of feature values. As each combination of feature values can be given an independent number of 0 and 1, we have \( 2^4 = 16 \) possible binary mapping functions with two binary inputs. In general, the possible combinations of feature values is \( n_c = b^n \), where \( b \) is the number of possible feature values (\( b = 2 \) for binary feature values) and \( n \) is the dimensionality of the feature space. The number of possible mapping functions is \( n_f = b^n_c \), which can be very large. Thus, the size of the table grows very fast with the number of feature components and the number of possible feature values. For example, we need a look-up table of size \( 2^{100} \approx 10^{30} \) for the recognition of letters binarized by our simple model retina, and the size increases with a power law when increasing the resolution of the digitization process. Moreover, the size is infinitely large if the feature values are real-valued. There are some fixes to the problem of the large sizes needed for look-up tables, such as not list all possible combinations of the feature values in the look-up table. This would make sense in the OCR because because there not all combinations correspond to a letter. We could simply ignore those combinations and have our system generate a ‘not a letter’ result if the combination is not found in the partial look-up table. However, the number of possible representations of the letters is still very large. Furthermore, we have to ask how to build such a look-up table. In our example we would have to generate all possible representations of letters, which seems unrealistic at least with respect to the development of our perceptive system.

Another possibility for realizing a mapping function, which has often been proposed as a model for human perception, is the utilization of prototypes. A prototype is a vector that encapsulates, on average, the features for each individual object. In our letter representation example we would need to store only 26 prototypes in a look-up table, one for each different letter. We have then to add another process that maps each sensory feature vector on to a specific prototype vector. One way to achieve this is to calculate the ‘distance’ of the sensory feature vector from each prototype and choose the prototype that is ‘closest’ to the sensory feature vector.

A remaining question in the prototype scheme is how to generate the prototype vectors. A possible scenario is to present a set of sample letters to the
system and to use the average (or another statistical measure) as a prototype for each individual letter. The interesting idea behind this scenario is that the generation of the prototypes is driven by examples. The disadvantage of the specific prototype scheme discussed here is that we have to compare each example to be recognized to each individual prototype vector. Thus, the computational requirement and time for recognition grows linearly with the number of prototypes.

### 6.1.3 The population node as perceptron

We demonstrate in this section that a population node, as introduced in Section 3.4, can represent certain types of vector functions. For this we set the firing rates of the input channels to

\[ r_i^{in} = x_i. \]  

(6.2)

The node in this simple perception system has to have at least \( n \) input channels, where \( n \) is the dimensionality of the feature space. For example, a simple perceptron in a two-dimensional feature space would look like the one illustrated in Fig. 6.3. The firing rate of the output represents a function

\[ \tilde{y} = r^{out}. \]  

(6.3)

To define the system completely we have to choose an activation function \( g \) for the node. We will start with the linear activation function \( g(x) = x \) and discuss others below. The output of a *linear perceptron* with two inputs is

\[ \tilde{y} = w_1 x_1 + w_2 x_2, \]  

(6.4)

where \( w_1 \) and \( w_2 \) are the weight values assigned to each input channel.

More generally, we can state the update rule of a single-layer mapping network with several output nodes, also called *simple perceptron*,

\[ r^{out} = g(w r^{in}), \]  

(6.5)

where \( g \) is an activation function. We used thereby a matrix notation (see eqn 4.20). This equation is equivalent to writing the update rule for each component,

\[ r_i^{out} = g(w_i x_1 + w_i x_2 + \ldots) = g(\sum_j w_{ij} r_j^{in}). \]  

(6.6)

Which functions \( \tilde{y} = r^{out} \) can be represented by simple perceptrons? To study this we return to the simple case of a linear population node and see if it can represent a particular function, for example, the function listed partially in the look-up table Table 6.2.

The first feature vector in this table is \( x^1 = (1, 2)' \). To calculate the value of the output, given the particular values of the input, we have to specify the weight values \( w_1 \) and \( w_2 \). We have the freedom to give these parameters any value we want. Let's choose the values \( w_1 = 1 \) and \( w_2 = -1 \). The output of the node is then

\[ \tilde{y}^1 = \tilde{y}(x^1) = \tilde{y}(x_1 = 1, x_2 = 2) = 1 \cdot 1 - 1 \cdot 2 = -1 = y^1. \]  

(6.7)
It is not surprising that we get, \( \bar{y}^1 = y^1 \), an exact match between the output of the node and the function value we want to represent. The reason for this is that we choose the weight values accordingly. One could achieve the same result with other values, for example, \( w_1 = -1 \) and \( w_2 = 0 \). Indeed, there are an infinite number of solutions to represent the function value \( y_1 \) because we have only one constraining equation with two free parameters. The reason that we chose \( w_1 = 1 \) and \( w_2 = -1 \) is that we are then also able to represent the second value in the look-up table,

\[
\bar{y}^2 = 1 \cdot 2 - 1 \cdot 1 = 1 = y^2.
\] (6.8)

At this stage, we have used up all free parameters of the weight vector \( w \) in order to represent the first two entries in the look-up table, and all the other values of the function \( \bar{y} \) are uniquely defined. The third entry of the look-up table is also correctly represented by the perceptron, namely

\[
\bar{y}^3 = 1 \cdot 3 - 1 \cdot (-2) = 5 = y^3.
\] (6.9)

We can say that the network generalized correctly. The reason for this match is that the third point lies on the two-dimensional sheet defined by eqn 6.4 as illustrated in Fig. 6.4. However, the fourth point in the look-up table, \( y^4 \), does not lie on this output sheet. Thus, the specific linear perceptron cannot not represent all the points of the look-up table in Table 6.2.

What about other activation functions? If we choose the activation function to be \( g(x) = \sin(x) \) instead of a linear function we are able to represent all four points listed in Table 6.2 (as can be verified easily). However, what about possible additional points not listed in the look-up table? For more complex functions we have to introduce increasingly complex activation functions in order to be able to represent them with a population node. However, an entire solution to the representation of mapping functions by a single node with a complex input–output relationship seems physiologically unrealistic. A more realistic solution is to use networks.

### 6.1.4 Boolean functions: the threshold node

Before leaving the single population node perceptron we briefly explore an important subset of the possible vector functions, the class of binary functions or Boolean functions. These are functions that have only two possible feature values for each feature component, which we choose to be \{0, 1\} in the following. This important set of functions represents only a small subset of all possible functions, but we know that we can build sophisticated (and indeed universal) learning machines from these basic functions. For Boolean functions it is natural to use a threshold function,

\[
g(x) = \begin{cases} 
1 & \text{if } x > \Theta \\
0 & \text{elsewhere} 
\end{cases},
\] (6.10)

as the activation function of a node since it limits the output values to the required binary values. The threshold node is equivalent to the McCulloch-Pitts neuron discussed in Section 3.1.6. A simple network of such threshold
units was originally termed ‘perceptron’ by Frank Rosenblatt and colleagues. We adapt here the term for more general feedforward networks with arbitrary activation functions to be consistent with later uses of this term when discussing multilayer versions.

A. Boolean OR function

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ w_1 x_1 + w_2 x_2 = \Theta \]

B. Boolean XOR function

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 6.5 (A) Look-up table, graphical representation, and single threshold population node with bias channel for the Boolean OR function. (B) Look-up table and graphical representation of the Boolean XOR function, which cannot be represented by a single threshold population node because this function is not linear separable. A node that can rotate the input space and has a non-monotonic activation function can, however, represent this Boolean function.

A simple threshold perceptron is able to represent a lot of Boolean functions. An example, the Boolean OR function, is illustrated in Fig. 6.5A. As shown in the look-up table on the left, the OR function is equal to $y = 1$ if either one of the inputs is one, and $y = 0$ otherwise. The middle graph represents this function in a graphical way by representing the binary values of $y$ with either a white or a black dot at the locations for the possible combinations of the input values $x_i$. This function can be represented by the single threshold population node with the parameters shown on the right in Fig. 6.5A. We typically included the threshold parameter as weight value $w_0$ for a channel with constant input. Such a bias input allows the threshold parameter to be included in the learning algorithms outlined below.

All but two out of the 16 possible Boolean functions with two input features can be represented by the threshold population node with the appropriate choice of weight values. The activation of the population nodes with two weight values and a possible constant bias defines a linear function, and the threshold activation function sets the output to the appropriate value as long as the linear activation curve can separate the two classes of $y$ values appropriately. We say that such classes of functions are linear separable. The only
two functions that cannot be represented are the XOR (exclusive-OR) function and its corresponding reverse, non-XOR. The XOR function is defined by the look-up table in Fig. 6.5B. The graphical representation reveals the reason for the failure, it is not possible to divide the regions of the feature values with different $y$-values with a single line that can be implemented by the activation function of the population node. We say that this function is not linear separable.

It does not seem too bad that only two out of 16 functions cannot be represented. However, the number of non-linear separable functions grows rapidly with the dimension of the feature space and soon outgrows the number of linear separable functions (see Table 6.3). The problem that only linear separable Boolean functions can be represented by a single threshold perceptron, often stated as the XOR-problem, greatly diminished the interest in simple perceptrons for several years until multilayer perceptrons (Section 6.2) became tractable. However, it is worth realizing that we can represent the XOR function with a single population node if we employ a non-linear activation function. For example, as illustrated on the right in Fig. 6.5B, we can change the representation of the problem by rotating the axis and using a simple perceptron with a Gaussian activation function. Such non-monotonic activation functions seem physiologically not plausible when we have a single neuron in mind, since stronger input to a neuron typically elicits stronger responses. However, as we stressed in Chapter 5, nodes are primarily intended to represent a collection of neurons, and we will discuss in the next chapter that single neurons have tuning functions resembling Gaussians.

### 6.1.5 Learning: the delta rule

Perceptrons have enormous potential for approximating functions by properly choosing values for the weights of the input channels. How can we choose appropriate values? This seems an enormous task given the vast number of adjustable parameters in large networks and the exploding number of their combinations. The solution to this problem, one of the great achievements and attractions of neural network computing, lies in algorithms that can find appropriate values from examples. The process of changing the weight values to represent the examples is often called learning, training, or adaptation. We
will usually use the term \textit{learning algorithms} for algorithms that adjust the weight values in abstract neural networks. Learning is an essential ingredient in forming memories and other cognitive functions, as discussed later in this book. If we are primarily interested in building learning machines, then we have no biological constraints and could discuss statistical learning in general terms. We will mention some general machine learning algorithms below. However, if we are interested in understanding biological learning and memory, then we need to relate such algorithms to physical changes in the brain, such as synaptic plasticity, discussed in Chapter 4.

In this section we show how simple mapping networks can be trained. Our objective in the following is to minimize the mean difference between the output of a feedforward mapping network and a desired state provided by a teacher. We can quantify our objective with an \textit{objective function} or \textit{cost function}, labelled $E$ in the following, which measures the distance between the actual output and the desired output. There are many possible definitions of a distance between two states. Formally, each definition defines the metric of the state space as long as the distance measure has certain obvious characteristics, such as that the distance is zero only if the output of the network is equal to the desired state, and the distance is positive otherwise. Some examples of distance measures are listed in Appendix A. A simple and often used cost function is the \textit{mean square error} (MSE)

$$ E = \frac{1}{2} \sum_i (r^\text{out}_i - y_i)^2, $$

where $r^\text{out}_i$ is the actual output and $y_i$ is the desired output of a mapping network. The factor $1/2$ is a useful convention when using the MSE as an objective function, as we will see below. If the output of each output node is equal to the desired value $y_i$ for each node, then this distance is zero. It is larger than zero when one or more of the components do not agree.

We can minimize the error function between the desired output and the actual output of a single-layer mapping network by changing the weight values to the output layer along the negative gradient of the error function,

$$ w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \quad \text{with} \quad \Delta w_{ij} = -\epsilon \frac{\partial E}{\partial w_{ij}}. $$

This is called a \textit{gradient descent} method. The error function is monotonically decreased by this procedure in steps proportional to the gradient along the error function. The constant $\epsilon$ is a \textit{learning rate}. An example is shown in Fig. 6.6 for a hypothetical network with only one weight value $w$. By starting with some random weight value for this 'network' it is likely that this network shows poor performance with a large error. The weight value is then changed in the direction of the negative gradient, which is a vector pointing in the direction of the maximal slope of the objective function from this point, and has a length proportional to the slope in this direction. Changing the weight values of the network to new values in this direction therefore guarantees that the network with the new weights has a smaller error than before. When approaching a minimum the gradient of the error function is decreasing and the change in the
weight values slows down. This method often results in a rapid initial decrease of the network error.

We can easily derive the particular algorithm to implement the gradient descent method for a simple perceptron when using the MSE as a measure of the distance between desired output $y_i$ and actual output $r_i^{\text{out}}$ of the perceptron. The gradient of the error function is given by

$$
\frac{\partial E}{\partial w_{ij}} = \frac{1}{2} \frac{\partial}{\partial w_{ij}} \sum_i (g(\sum_j w_{ij}r_j^{\text{in}}) - y_i)^2 
$$

$$
= f'(h_i)(\sum_j w_{ij}r_j^{\text{in}}) - y_i)r_j^{\text{in}}. 
$$

(6.13)

$f'(x) = \frac{\partial f(x)}{\partial x}$ is the derivative of the activation function that enters the formula by using the chain rule for calculating differentials

$$
\frac{df(g(x))}{dx} = \frac{df}{dg} \frac{dg(x)}{dx}. 
$$

(6.14)

The resulting learning rule,

$$
w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \quad \text{with} \quad \Delta w_{ij} = \epsilon g'(h_i) \ast (y_i - r_i^{\text{out}}) r_j^{\text{in}},
$$

(6.15)

is called the delta rule as the change of the weight values is proportional to the difference between the desired and actual output often expressed as $\delta$-term, $\delta = y_i - r_i^{\text{out}}$. For a linear perceptron with activation function $f(x) = x$, this is simply given by $f' = 1$. The weight change for a linear perceptron is therefore

$$
\Delta w_{ij} = \epsilon(y_i - r_i^{\text{out}}) r_j^{\text{in}}. 
$$

(6.16)

This rule, without the derivative of the activation function, was used previously for threshold perceptrons and is called the perceptron learning rule. Since the threshold function is not differentiable at the threshold, this function is formally not a gradient descent rule. However, it turns out that this rule still works for training the threshold perceptron on simple classification problems.

Also, note the similarity of the delta learning rule with Hebbian plasticity as discussed in Chapter 4 (eqn 4.10). This learning rule has now two Hebbian terms. The weights are increased with supervision by an amount proportional to the product of the presynaptic node (input value) $r_j^{\text{in}}$ and the desired postsynaptic value $y_i$. This is a supervised Hebbian learning term. However, the weights are also decreased by the product of the input value $r_j^{\text{in}}$ and the actual postsynaptic value $r_i^{\text{out}}$. This term is like unlearning the actual response of the perceptron when the perceptron does not give the right answer. Learning ceases when the actual output is equal to the desired output, since the $\delta$-term is then zero. The algorithm of the delta rule is summarized with general activation functions in Table 6.4.

Simulation

Digitized versions of the alphabet are provided in the text (ASCII) file pattern1 in folder Chapter6. The two states are coded with 0s and 1s. This file can be
Table 6.4 Summary of delta-rule algorithm

Initialize weights arbitrarily
Repeat until error is sufficiently small
  Apply a sample pattern to the input nodes: \( r_i^\text{in} = \xi_i \)
  Calculate rate of the output nodes: \( r_i^\text{out} = g(\sum_j w_{ij} r_j^\text{in}) \)
  Compute the delta term for the output layer: \( \delta_i = g'(h_i^\text{out})(c_i^\text{out} - r_i^\text{out}) \)
  Update the weight matrix by adding the term: \( \Delta w_{ij} = \epsilon \delta_i r_j^\text{in} \)

loaded in the MATLAB editor to inspect the letters. Alternatively, individual letters can be displayed with the function `displayLetter` shown in Table 6.5. The `load` command in Line 3 reads the content of file `pattern1` into an array with the same name. The MATLAB function `reshape()` in Line 4 reshapes this \( 312 \times 13 \) matrix to a \( 156 \times 26 \) matrix with a column vector for each letter in the alphabet. This is useful as we can then easily select one of the letters, as done in Line 5. The letter vectors can be used as input vectors to the perceptron. Of course, for humans to recognize the letter on screen, we need to reshape the letter vector again into an array as done in Line 6, before displaying it in Line 7 with a special output format. An example output of this function is shown in Fig. 6.7A.

Table 6.5 Function `displayLetter.m`

```matlab
1 function displayLetter(n);
2 % Displays letter number n from file pattern1
3 load pattern1
4 letterVectors=reshape(pattern1', 12*13, 26);
5 thisLetterVector=letterVectors(:,n);
6 thisLetterArray=reshape(thisLetterVector, 13, 26)';
7 format +; disp(thisLetterArray); format;
8 return
```

File `perceptronTrain.m`, shown in Table 6.6, contains a training program for a single-layer perceptron. The weight matrix \( w_{\text{out}} \) is initially set to random values with positive and negative components in Line 4. In Line 8, the `reshape()` function is used as in function `displayLetter` to create a pattern matrix \( r_{\text{In}} \) used as inputs for the perceptron. We choose a distributed representation as desired output by creating a diagonal matrix in Line 9, so that an \( A \) is represented by a vector with a one as first component, and zeros elsewhere, etc.

We then loop over several training steps in which all patterns are presented to the network. In Line 14, we calculate the actual output of the network. As we supply all the patterns vectors at once (input pattern matrix), the output is
A. Training pattern

```
>> displayLetter(1)
  
+++ 
+*++
++*
++++++
++
++++
+++ 
+++ 
+++ 
```

B. Learning curve

![Learning curve graph]

C. Generalization ability

![Generalization ability graph]

Fig. 6.7 (A) Example of using the function displayLetter(n) to plot the content of file pattern1. (B) Example of a training function during training a threshold perceptron on the letter patterns with the delta rule. (C) Performance of the trained perceptron with noisy versions of the original pattern. When interpreting the output after training, two different activation functions are shown.

Also a matrix with an output vector for each input vector. The calculation of the output includes the threshold activation function. The training is done with the perceptron learning rule. The performance of the recognition is tested before each training step, using as error measure the average number of different bits (Hamming distance) between the actual and the desired output. An example training of a curve produced with the program is shown in Fig. 6.7B. The curve starts with roughly half the bits wrong in the prediction of the untrained network. This recognition error rapidly decreases and the network is fully trained after 18 training steps, when all input vectors create the desired output.

Just learning (memorizing) specific patterns is not our only goal, we also want to test the generalization ability of the trained network. We do this in program perceptronTest shown in Table 6.7. The training is first run in Line 2, and the instruction of the next line saves the original letter matrix in matrix letterMatrix. We then test the performance when flipping a specific number of bits at random positions in each letter vector. To do this, we create a matrix with function randomFlipMatrix() shown in Table 6.8 with zero components except in nflip random positions in each column vector. In this function, we use the MATLAB function randperm(n) to generate a random permutation of the numbers 1 to n. The first nflip numbers of this sequence can be used as positions at which the components in a column vector are set to one. The resulting matrix is then subtracted from the original letter matrix, which results in flipping the bits at the positions with ones in randomFlipMatrix when taking the absolute value of this difference.

The test program goes on to test the noisy patterns by calculating how many letters are wrongly recognized in Lines 10, 11 and 16. These results are averaged over 10 trials for each number of flipped bits since the random process of flipping can produce different results. The program also demonstrates how to create a concatenated vector of results in Lines 12 and 17, how to calculate means and standard deviations of the components in the resulting vector using MATLAB functions in Lines 18 and 19, and how to produce a plot with error
bars in Lines 22 and 23.

The test is done in this program for two different activation functions of this trained network. The first one is the threshold function which was used for training. The results are shown in Fig. 6.7C in the upper curve. The results show some patterns can not be recalled even with small amounts of noise, while others can. Of course, the recognition does deteriorate until all letter recognition is lost at around a noise level of 20%. There are many ways to improve the recognition ability of such networks. For example, we demonstrate here the use of an activation function that activates only the node with the maximal net input. The result of this activation function is also shown in Fig. 6.7C. Now all letters are robust to some amount of noise, and the network can tolerate more noise. A biological implementation of the maximum function can be achieved with lateral inhibition where the most active node tends to suppress other nodes. We follow up this line of thought in the next chapter.

6.2 The multilayer perceptron

We have seen that the limited number of weight values in a single neuron limits the complexity of functions that we can represent using a single population
Table 6.7 Program perceptronTest.m

```matlab
1 8% Testing generalization performance of trained perceptron
2    perceptronTrain;
3    letterMatrix=rIn;
4    for nflip = 1:80;
5        dist1=[]; dist2=[];
6        for trial=1:10;
7            rIn=abs(letterMatrix-randomFlipMatrix(nflip));
8            % Threshold output function
9            rOut1=(wOut*rIn)>0;
10           nerror=0;
11            for j=1:26; nerror=nerror+(sum(rDes(:,j)==rOut1(:,j))>0); end
12            dist1=[dist1,nerror];
13            % Max output function
14            [v,i]=max(wOut*rIn);
15            rOut2=zeros(26); for j=1:26; rOut2(i,j)=1; end
16            dist2=[dist2,0.5*sum(sum(rDes==rOut2))];
17        end
18        meanDist1(nflip)=mean(dist1); stdDist1(nflip)=std(dist1);
19        meanDist2(nflip)=mean(dist2); stdDist2(nflip)=std(dist2);
20    end
21    figure; hold on;
22    errorbar((1:80)/156,meanDist1,stdDist1,'.');
23    errorbar((1:80)/156,meanDist2,stdDist2,'r');
24    xlabel('Fraction of flipped bits')
25    ylabel('Average number of wrong letters')
```

node. An obvious solution therefore is to increase the number of nodes, and thereby the number of connections with corresponding independent weight values. However, by doing so we want to keep the number of input channels, representing the feature values, and the number of output channels, representing the dimension of the internal object representation, constant. The only solution is to use hidden nodes. An example of such a multilayer mapping network, often called multilayer perceptron, is illustrated in Fig. 6.8. The middle layer is commonly called a hidden layer as the nodes in this layer have no direct input or output channel to the external world.

The number of weight values, \( n^w \), in networks with a hidden layer grows rapidly with the number of nodes, \( n^h \), in the hidden layer,

\[
    n^w = n^{in}n^h + n^h n^{out},
\]

where \( n^{in} \) is the number of input nodes and \( n^{out} \) is the number of output nodes. We have neglected the weights of the input channels because we merely assigned to the input nodes the role of distributing the input value to all of the other nodes. These nodes are hence not directly computing nodes, and we adopt the common nomenclature and call the architecture illustrated in Fig. 6.8 a
6.2 The multilayer perceptron

Table 6.8 Function randomFlipMatrix.m

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>function r=randomFlipMatrix(n);</td>
</tr>
<tr>
<td>2</td>
<td>% returns matrix with components 1 at n random positions of each column</td>
</tr>
<tr>
<td>3</td>
<td>r=zeros(156,26);</td>
</tr>
<tr>
<td>4</td>
<td>for i=1:26</td>
</tr>
<tr>
<td>5</td>
<td>x=randperm(156);</td>
</tr>
<tr>
<td>6</td>
<td>r(x(1:n),i)=1;</td>
</tr>
<tr>
<td>7</td>
<td>end</td>
</tr>
</tbody>
</table>

\[ n^{in} \rightarrow r^h \rightarrow n^{out} \]

Fig. 6.8 The standard architecture of a feedforward multilayer network with one hidden layer, in which input values are distributed to all hidden nodes with weighting factors summarized in the weight matrix \( w^h \). The output values of the nodes of the hidden layer are passed to the output layer, again scaled by the values of the connection strength as specified by the elements in the weight matrix \( w^{out} \). The parameters shown at the top, \( n^{in} \), \( n^h \), and \( n^{out} \), specify the number of nodes in each layer, respectively.

\textit{two-layer} network. It is straightforward to include more layers of hidden nodes as discussed later.

6.2.1 The update rule for multilayer perceptrons

With the vector and matrix notation mentioned before, we can state the functionality of a multilayer perceptron in a compact form, directly generalizing the update rule for a single-layer mapping network (eqn 6.5). An input vector is weighted by the weights to the hidden layer, \( w^h \), and all the inputs to one node are summed up. This corresponds to a matrix-vector multiplication,

\[
h^h = w^h r^{in}, \tag{6.18}\]

which is a compact way of writing all the equations for the individual components

\[
h^h_i = \sum_j w^h_{ij} r^{in}_j. \tag{6.19}\]

The vector \( h^h \) is called the activation vector of the hidden nodes. We next pass the activations of the hidden nodes through an activation function for
each node, which results in a vector that represents the population rates of the hidden layer,

$$r^h = g^h(h^h).$$  \hspace{1cm} (6.20)

This rate vector becomes the input vector to the next layer, which can be the output layer or another hidden layer in the case of a multilayer network with more than one hidden layer. The final output vector is calculated from the outputs of the last hidden layer,

$$r^{out} = g^{out}(w^{out}r^h).$$  \hspace{1cm} (6.21)

We can summarize all the steps in updating the states of the multilayer feed-forward network in one equation, namely

$$r^{out} = g^{out}(w^{out}g^h(w^h \cdot r^{in})).$$  \hspace{1cm} (6.22)

It is easy to include more hidden layers in this formula. For example, the operation rule for a four-layer network with three hidden layers and one output layer can be written as

$$r^{out} = g^{out}(w^{out}g^h(w^h g^{h2}(w^{h1}r^{in}))).$$  \hspace{1cm} (6.23)

This formula looks lengthy, but it is a straightforward generalization of the two-layer network and can be easily implemented with a computer program.

Let us discuss a special case of a multilayer mapping network where all the nodes in all hidden layers have linear activation functions ($g(x) = x$). Eqn 6.23 then simplifies to

$$r^{out} = g^{out}(w^{out}w^{h3}w^{h2}w^{h1}r^{in})$$

$$= g^{out}(\tilde{w}r^{in}).$$  \hspace{1cm} (6.24)

In the last step we have used the fact that the multiplication of a series of matrices simply yields another matrix, which we labelled $\tilde{w}$. Eqn 6.24 represents a single-layer network as discussed before. It is therefore essential to include non-linear activation functions, at least in the hidden layers, to make possible the advantages of hidden layers that we are about to discuss. We could also include connections between different hidden layers, not just between consecutive layers as shown in Fig. 6.8, but the basic layered structure is sufficient for the following discussions.

Which functions can be approximated by multilayer perceptrons? The answer is, in principle, any. A multilayer feedforward network is a universal function approximator. This means that, given enough hidden nodes, any mapping functions can be approximated with arbitrary precision by these networks. The remaining problems are to know how many hidden nodes we need, and to find the right weight values. Also, the general approximator characteristics does not tell us if it is better to use more hidden layers or just to increase the number of nodes in one hidden layer. These are important concerns for practical engineering applications of those networks. Unfortunately, an answer to these questions cannot be given in general. It depends on the function we want to approximate.
A network with threshold nodes that can represent the XOR function is shown in Fig. 6.9. How well a particular feedforward network can approximate a function depends on all network details, including the activation functions of the nodes. The most commonly used activation function in these networks is the sigmoid, or logistic, function,

\[
g(x) = \frac{1}{1 + e^{-\beta(x-x_0)}} = \frac{1}{2}(\tanh(\beta(x-x_0)) + 1),
\]

where the parameter \( \beta \) controls the slope, and \( x_0 \) is the offset value, where the function has the largest slope and is equal to \( g(x_0) = 1/2 \). This particular function has contributed greatly to the success of such networks in technical applications. A partial reason for this is that the sigmoid function has the useful characteristic that it changes smoothly in a somewhat confined area, and is nearly constant outside this area. By combining several such sigmoid functions we can quickly generate complex functions. We can adjust the approximation locally by adding a sigmoid function to a given approximation and subtracting a second sigmoid function with a slightly different offset. An example of approximating a sine function by the sum of a small number of sigmoid functions is illustrated in Fig. 6.10. Three dotted lines are drawn corresponding to sigmoidal activation functions of hidden nodes with the same slope and amplitude but different offsets. The sum of these functions, as can be calculated by a linear output node, is shown as a dashed line. The dashed line approximates one period of the sine function reasonably well, although the approximation is quite bad outside of this central region.

### 6.2.2 Generalization

An important aspect of regression models is their generalization ability. Generalization refers to the performance of the network on data that were not part of the training set. For example, to approximate the sine function in Fig. 6.10 with three sigmoidal nodes, we need only a few few training examples to fix the parameters of the three sigmoidal functions to fit the sine function reasonably well. The training points are usually well approximated by the network, and points in between the training points are also fairly well approximated by the network. The generalization ability of interpolation with sigmoidal networks is often quite good when the true function does not vary widely between training points. In contrast, the generalization ability of extrapolation with such networks into unseen domains is often poor. The reason for this is that sigmoidal networks basically assume a constant dependence for such data points that might not be appropriate, as in the example shown in Fig. 6.10.

To enable good generalization abilities of networks, we need to consider carefully the degree of freedom of the function approximator, which is usually directly related to the number of free parameters that have to be determined by the learning procedure. The problem is illustrated in Fig. 6.11. We have chosen there a training set of six data points (shown as stars) derived from the 'true' function \( f(x) = 1 - \tanh(x-1) \) (shown as a solid line) to which a small random number was added, values which might represent noise in the measurements. A large network, which has many free parameters, will be able...
to fit the training data with high precision but shows high variance between these points (dotted line). This is called \textit{overfitting}. In contrast, a model with too few parameters, such as the straight dashed line, will have a high bias. This is called \textit{underfitting}.

Several methods have been proposed to prevent over- or under-fitting. For example, high variance in a model with many parameters is often only seen late in training when the training algorithm tries to approximate the training examples with an unreasonable precision. \textit{Early stopping} of the training has therefore been recommended to combat overfitting. Many researchers also like to restrict the number of hidden nodes to limit the number of parameters.

The problem of a \textit{gain-variance trade-off} can be tamed more systematically with a method called \textit{regularization}. This method is often used by introducing constraints on the number of parameters in the cost function that should be minimized by the learning rule. For example, an old idea for neural networks was to restrict the weight values by penalizing large weights. To realize this, we can add a term proportional to the length of weight vector to the MSE function used earlier,

\begin{equation}
E = \frac{1}{2} \sum_i (r_i^{\text{out}} - y_i)^2 - \gamma r \frac{1}{2} \sum_i w_i^2,
\end{equation}

where $\gamma$ sets the relative importance of this regularization term relative to the estimate of the generalization error. A gradient descent on this cost function leads to a learning rule with a term resembling weight decay. We will see in the last section of this chapter that this form of regularization can be given a solid theoretical underpinning with large \textit{margin classifiers} and \textit{structural risk minimization}. Regularization is therefore an essential ingredient in the proper use of machine learning methods. However, the following derivation of the standard gradient descent learning rule is, for simplicity, based on MSE alone.

\subsection{The generalized delta rules}

Learning has only been discussed for single-layer networks in Section 6.1.5. The delta rule (eqn 6.15) cannot be applied directly to each layer of a multilayer mapping network since this would require a teaching signal for each node in the
network. The desired value of the output node can be supplied by a teacher, but proper values of the hidden nodes are not known a priori. It is, however, straightforward to generalize the delta rule in the gradient descent formalism. It is therefore surprising that this generalization was widely recognized only from the mid 1980s onward, although the algorithm was known and used for training neural networks long before. Let us illustrate the algorithm for a multilayer feedforward network with one hidden layer denoted by the superscript ‘h’. The gradient of the MSE error function with respect to the output weights is given by

\[
\frac{\partial E}{\partial w_{ij}^{\text{out}}} = \frac{1}{2} \frac{\partial}{\partial w_{ij}^{\text{out}}} \sum_i (r_i^{\text{out}} - y_i)^2 \\
= \frac{1}{2} \frac{\partial}{\partial w_{ij}^{\text{out}}} \sum_i (g^{\text{out}}(\sum_j w_{ij}^{\text{out}} r_j^{\text{h}}) - y_i)^2 \\
= g'_{\text{out}}(h_i^{\text{h}})(\sum_j w_{ij}^{\text{out}} r_j^{\text{h}} - y_i) r_j^{\text{h}} \\
= \delta_i^{\text{out}, h_j},
\]

(6.27)

where we have defined the delta factor

\[
\delta_i^{\text{out}} = g'_{\text{out}}(h_i^{\text{h}})(\sum_j w_{ij}^{\text{out}} r_j^{\text{h}} - y_i) \\
= g'_{\text{out}}(h_i^{\text{h}})(r_i^{\text{out}} - y_i).
\]

(6.28)

Eqn 6.27 is just the delta rule as before because we have only considered the output layer. The calculation of the gradients with respect to the weights to the hidden layer again requires the chain rule as they are more embedded in the error function. Thus we have to calculate the derivative

\[
\frac{\partial E}{\partial w_{ij}^{\text{h}}} = \frac{1}{2} \frac{\partial}{\partial w_{ij}^{\text{h}}} \sum_i (r_i^{\text{out}} - y_i)^2 \\
= \frac{1}{2} \frac{\partial}{\partial w_{ij}^{\text{h}}} \sum_i (g^{\text{out}}(\sum_j w_{ij}^{\text{out}} g^{\text{h}}(\sum_k w_{jk}^{\text{h}} r_k^{\text{in}})) - y_i)^2.
\]

(6.29)

After some battle with indices (which can easily be avoided with analytical calculation programs such as MAPLE or MATHEMATICA), we can write the derivative in a form similar to that of the derivative of the output layer, namely

\[
\frac{\partial E}{\partial w_{ij}^{\text{h}}} = \delta_i^{\text{h}} r_j^{\text{in}},
\]

(6.30)

when we define the delta term of the hidden term as

\[
\delta_i^{\text{h}} = g'(h_i^{\text{in}}) \sum_k w_{ik}^{\text{out}} \delta_k^{\text{out}}.
\]

(6.31)

The error term \(\delta_i^{\text{h}}\) is calculated from the error term of the output layer with a formula that looks similar to the general update formula of the network, except that a signal is propagating from the output layer to the previous layer. This is the reason that the algorithm is called the error-back-propagation algorithm. The algorithm is summarized in Table 6.9.
Table 6.9 Summary of error-back-propagation algorithm

Initialize weights arbitrarily
Repeat until error is sufficiently small
Apply a sample pattern to the input nodes: \( r_i^0 := r_i^{\text{in}} = x_i^{\text{in}} \)
Propagate input through the network by calculating the rates of nodes in successive layers \( l \): \( r_i^l = g(h_i^l) = g(\sum_j w_{ij}^l r_j^{l-1}) \)
Compute the delta term for the output layer:
\[
\delta_i^{\text{out}} = g'(h_i^{\text{out}})(x_i^{\text{out}} - r_i^{\text{out}})
\]
Back-propagate delta terms through the network:
\[
\delta_i^{l-1} = g'(h_i^{l-1}) \sum_j w_{ij}^l \delta_j^l
\]
Update weight matrix by adding the term: \( \Delta w_{ij}^l = \alpha \delta_i^l r_j^{l-1} \)

Simulation

The implementation of the error-back-propagation algorithm is demonstrated with the program listed in Table 6.10. This program trains a sigmoidal network with two hidden nodes (\( N_h = 2 \) in Line 3) and one output node (\( N_o = 1 \)) on the XOR function discussed in Section 6.1.4, which has two input values (\( N_i = 2 \)). The weight matrices are initialized with small random values between -0.5 and 0.5 in Line 4. The training vectors are defined with the inputs in Line 7 and the desired output in Line 8. These training patterns represent the XOR function.

The training over 10,000 training examples is started with the loop in Line 11. In each of these training steps, one of the four possible patterns is selected (Line 13). This pattern is propagated through the network by calculating the activation of the hidden nodes on Line 14, and the activation of the output node on Line 15. The \( \delta \)-terms (see Table 6.9) for the output layer and the hidden layer are calculated in Lines 16 and 17, and then used for the weight updates with the delta rule in Lines 18 and 19. After this, all the patterns are tested by propagating them through the network in Line 21 with the new weight matrices, and half of the summed squared distance between the actual output and the desired output is recorded in Line 22. Half of the mean square distance would be \( 1/4 \) of this value. The learning curve, which is the development of the training error, is plotted in Line 24.

A typical training curve produced with this program is shown in Fig. 6.12 (each time the program is called a different learning curve is produced because different random initial weight values are used and a different order of the training examples is presented during learning). Due to the small initial weights, the network always responds initially with values around \( r^{\text{out}} \approx 0.5 \) which makes the mean square error relatively small from the start. It then takes many iterations, even with the relatively large learning rate of 0.7, to finally reduce the error significantly and to reach a level of performance that is sufficient to represent the function correctly. The network responded after the 10,000 learning steps shown in the example with the values.
to the four different training vectors. Such a response pattern is sufficient to represent the XOR function. Just imagine that we use a final threshold output node with threshold value 0.6.

The basic error-back-propagation is known to be very inefficient in training neural networks, and many advances have been made to improve the performance considerably, some of which we mention below. MATLAB provides a neural network toolbox that implements many such advanced algorithms, and shareware versions of different algorithms can also be found on the web. In particular, the neural network package written by Ian Nabney and Christopher Bishop called Netlab (see http://www.ncrg.aston.ac.uk/netlab) is very useful and has many advanced implementations.

6.2.4 Biological plausibility of MLPs

While mapping networks are an important ingredient in brain processing, and while MLPs have been very useful in understanding principal issues in statis-
tical learning theory, we must advise caution when applying such networks to brain theories. We mentioned above that feedforward mapping networks are universal function approximators, which necessitates caution when interpreting them in computational neuroscience. For example, in experiments we often measure some response function as a function of some parameters controlled experimentally. A multilayer mapping network, as universal approximator, will be able to approximate every such function arbitrarily well with the right choice of parameters in the network. When such networks are used to fit experimental data one might be tempted to claim that these systems represent models of the brain, on the basis that the processing nodes in the network resemble neurons. Such claims have contributed to the low acceptance of such models in the neuroscience community.

Network approximation of a particular problem depends strongly on the number of hidden nodes and the number of layers. Adding more hidden nodes can drastically change the representation of learning examples in the network. An interpretation of hidden node activities in such networks is therefore questionable when the models are aimed at a neuronal level. In connectionist models it is common to limit the number of hidden nodes drastically to enable better generalization performances of such networks. Using a small number of hidden nodes results in a relatively smooth interpolation between training data, and a better generalization can be expected for smooth problems. However, the number of hidden nodes in biological systems can be very large, which makes this kind of analysis different from building actual brain models on a neuronal level.

Training MLPs with the generalized delta rule (error-back-propagation) is particularly problematic for biological interpretations, specifically if this training is intended to model the dynamics of biological learning based on synaptic plasticity. The back-propagation of error signals, necessary for the generalized delta rule, seems difficult to realize in cortical networks. While some form of information exchange between postsynaptic and presynaptic neurons is possible, the wide use of such mechanisms for a back-propagation of errors through the whole network introduces several other problems. A major problem is the nonlocality of the algorithm in which a neuron has to gather the back-propagated errors from all the other nodes to which it projects. This not only raises synchronization issues, but also has disadvantages for true parallel processing in the system. The inclusion of derivative terms in the delta signals is also problematic. The back-propagation of inaccurate derivative terms can quickly lead to inaccurate updates of the weights in the network. Finally, it has never been resolved how a forward propagating phase of signals can be separated effectively from the back-propagation phase of the error signals. Some alternative schemes closely resembling the basic scheme of error-back-propagation, while claiming to be more biologically realistic, have been proposed in the literature, although direct verifications have not yet been established.

The caution that must be used in applying mapping networks to brain modeling does not negate the importance of mapping networks in the brain. Information is frequently mapped between different representations, and cognitive abilities are supported by such mapping abilities. A better understanding of precisely how this is achieved in the brain is a major topic in computational
6.3 Advanced MLP concepts

6.3.1 Kernel machines and radial-basis function networks

The activity of a node in a neural network is determined by the inner product between an input vector \( \mathbf{r} \) and the weight vector of the node, \( \mathbf{w} \). With our convention that the weight vector is a line vector and the input feature vector is column vector, we can write the net input of a node as

\[
h = \mathbf{wr} = \sum_i w_i r_i. \tag{6.32}
\]

In the letter recognition example, we used the value of each pixel of the 'retinal' image, which we label here as \( x_i \), for each component of the feature vector \( r_i \). In machine learning it is common to call such basic features, which are given to us, the attributes. It has long been recognized that better performance of networks can sometimes be achieved by including combinations of attributes, such as \( x_1 x_2 \) or \( x_1^2 \), as features in the machine learning task. We can write such a transformation of the original feature space to the new feature space as

\[
\mathbf{x} \rightarrow \phi(\mathbf{x}) \tag{6.33}
\]

The new feature vector has, of course, a higher dimensionality, so that we also need to use more weight values. Since these weight values are anyhow parameters that we have to fix through a learning procedure, we can also introduce new parameters through the same transformation,

\[
\mathbf{w} \rightarrow \phi(\mathbf{w}). \tag{6.34}
\]

The node in the network for the transformed problem can then be written as

\[
\hat{h} = \phi(\mathbf{w})\phi(\mathbf{r}) = K(\mathbf{w}, \mathbf{r}). \tag{6.35}
\]

We have introduced thereby a new function \( K \), called the kernel function. This function calculates the scalar product of the transformed vectors. For example, let us consider a two-dimensional attribute vector \( (x_1, x_2)' \) and the
transformation that considers all combinations of the components,
\[
x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow \phi(x) = \begin{pmatrix} x_1 x_1 \\ x_1 x_2 \\ x_2 x_1 \\ x_2 x_2 \end{pmatrix}, \tag{6.36}
\]
which is a four-dimensional feature vector. The scalar product of \(\phi(w)\phi(x)\) can be calculated more easily as \((wx)^2\), as can easily be verified. We can also go the other way and propose a kernel function and then try to determine the corresponding feature space transformation. We are here particularly interested in the Gaussian kernel function
\[
K(w, x) = e^{-\frac{(w-x)^2}{2\sigma^2}} \tag{6.37}
\]
This kernel function is interesting as it fits well the shape of some response functions of neurons, as discussed more in Chapter 7. While the dimensionality of the corresponding transformed feature space is infinite, we do not have to calculate this transformation at all for the network, we only need to replace the original network (eqn 6.32) with the kernel machine (eqn 6.35).

Feedforward mapping networks with activation functions that depend on the distance between an input value and weight value, most commonly in the form of the Gaussian kernel eqn 6.37, are also called radial-basis function (RBF) networks. A typical RBF network is shown in Fig. 6.13. This network has three input nodes and four hidden nodes with a RBF activation function as shown in Fig. 6.14. Thus, the weights to the RBF nodes, \(c\), represent the centres of the radial basis functions. We will relate such networks to cortical maps with tuning curves in Chapter 7. The example shown has one output node with a linear activation function. Such an output perceptron can decode the feature representation in the hidden layer, as discussed further in Chapter 7.

### 6.3.2 Advanced learning

The basic error-back-propagation algorithm has many performance problems, and a lot of effort has been devoted to the improvement of the basic version of this gradient descent method. Improved techniques include the smart choice of initial conditions, different error functions, various acceleration techniques, and hybrid methods. We will only provide a short overview of some of these techniques without many details because they are often of minor biological significance and the subject of more technologically oriented publications.

In applications of the basic gradient descent method we can typically find an initial phase where the average error over the training examples is rapidly decreasing. However, this is unfortunately often followed by a phase of very slow convergence, often caused by a shallow part of the error function. Many solutions have been proposed to overcome this problem. One of the oldest is to use a \textit{momentum} term that 'remembers' the change of the weights in the previous time step,
\[
\Delta w_{ij}(t+1) = -\frac{\partial E}{\partial w_{ij}} + \alpha \Delta w_{ij}(t). \tag{6.38}
\]
The momentum term has the effect of biasing the direction of the new update vector towards the previous direction (hence the name ‘momentum’), which is often a good guess for improved weight values. Another method is to increase the learning rate when the gradient becomes small. Several methods with adaptive learning rates were proposed and are used in technical applications. Acceleration of the gradient descent method is very important as the convergence to an acceptable error level can take thousands of learning steps.

Shallow areas in the error function depend on the particular choice of the error function on which the gradient descent method is based. An acceleration of the learning process can often be achieved with error functions other than the MSE. A particularly interesting choice is the **entropic error function**,

\[
E = \frac{1}{2} \sum_{\mu,i} \left[ (1 + y_i^{(\mu)}) \log \frac{1 + y_i^{(\mu)}}{1 + r_{i}^{\text{out}}} + (1 - y_i^{(\mu)}) \log \frac{1 - y_i^{(\mu)}}{1 - r_i^{\text{out}}} \right],
\]

which is a measure for the information content (or entropy) of the actual output of the multilayer perceptron given the knowledge of the correct output. Although this error function looks computationally more demanding than the MSE, the application in gradient descent methods can be computationally less demanding. For example, the delta term of the output layer with an activation function \( g(x) = \tanh(x) \) reduces to \( \delta_i = y_i - r_{i}^{\text{out}} \). The appropriate choice of the error function depends on the statistical nature of the data. For example, it can be shown that the maximum likelihood estimate of Gaussian data correspond to minimizing the MSE error function. Modern machine learning has shed new light into this area.

The basic line search algorithm of gradient (or **steepest**) decent is known for its poor performance with shallow error functions. However, the training algorithms discussed here are based on the minimization of an error function, and we can employ many other advanced minimization techniques to achieve this goal. Several of these techniques, common in technical minimization procedures, take higher-order gradient terms into account. These can be viewed as including curvature terms describing the curvature of the error surface in the weight change calculations. Such methods typically involve the calculation of the inverse of the Hessian matrix, which can be numerically time-consuming. In the context of statistical learning theory we have to evaluate a proper distance measure such as the **Fisher information**. A superior gradient method, called **natural gradient algorithm**, has been proposed by Shun-ichi Amari based on such considerations.

The standard **Levenberg-Marquardt method** is also based on higher-order gradient information, and has been used to train feedforward mapping networks. This method is included in the MATLAB Neural Networks toolbox from The MathWorks, Inc. Simulations of training mapping networks with the natural gradient method and with the Levenberg-Marquardt method have demonstrated that these methods can overcome the problems of shallow minima (and even turning points in the error function), and that the convergence times for training the networks to a specified training error are many orders of magnitude less than those of other learning methods. The drastically improved convergence times can outnumber the increased simulation time due to the increased algorithmic complexity. Such algorithms should therefore be con-
sidered when applying mapping networks to technical problems. The relations of such algorithms to biological learning are, however, so far unclear.

A general limitation of pure gradient descent methods is the possibility that the network gets trapped in a local minimum of the error surface. The system is then not able to approach a global minimum of the error function as anticipated. This is illustrated in Fig. 6.6 where the global minimum on the right was actually not reached. A solution to this problem is to include some stochastic processes that enable random search. Several methods with random components are very successful in training multilayer perceptrons, most notably a method called simulated annealing. This method adds some noise to the weight values during the update of the weights on top of a deterministic algorithm such as gradient descent. This noise helps to escape shallow local minima. The noise level is then gradually reduced to ensure convergence.

A variety of methods utilize the rapid initial convergence of the gradient descent method and combine it with global search strategies. For example, we can employ a method in which, after the gradient descent method slows down below an acceptable level, a new starting point is chosen randomly. From this new starting point in the configuration space a few gradient descent steps are performed on the network. Only when the error value after these few gradient descent steps is lower than that at the previous level do we accept these new weight values for further gradient descent steps. Such hybrid methods combine the efficient local optimization capabilities of gradient descent methods with the global search abilities of stochastic processes. Genetic algorithms, discussed briefly below, use similar combinations of deterministic minimization and stochastic components.

6.3.3 Batch versus online algorithm

In technical applications of supervised learning, in which we have a set of examples of a function we want to represent with the mapping network, we can do the repetitions mentioned in the summary (Table 6.4, item 6) in various ways. A pattern can be learned very accurately with the gradient descent method if we train the network only on one pattern for a long time before switching to the next pattern. However, when a second pattern is trained consecutively in a similar way, then the network unlearns the previous pattern because the weights are adjusted entirely to represent the new pattern. This is not, of course, what we intended. In practical applications of the delta rule it is often (though not always) the best strategy to represent all patterns in a training set in a way that is known as a batch algorithm. In this strategy we first apply all patterns in a training set to the network and calculate the desired changes for each pattern. Only after all the patterns have been applied do we change the weights according to the mean value over all patterns before we repeat this procedure. Batch algorithms often work well in practice if there are not many redundancies in the training data, though batch algorithms are a bit more prone to get stuck in local minima.

However, batch algorithms are not plausible in biological systems, and it is more realistic to assume that the sensory information is processed directly. Online learning algorithms are therefore very important in computational neu-
These algorithms seem at first more restrictive and not as effective as batch algorithms as we can not repeatedly utilize past experiences. However, batch algorithms also have severe problems when applying them to many realistic applications. This becomes apparent if we think about training a network on realistic data, for example, on visual images. The data stream generated by the eyes is enormous, and even in artificial systems, such as systems that employ a digital camera with a much lower resolution than that of human eyes, the amount of data generated is still enormous. The amount of storage necessary to keep the images present for the batch algorithm will rapidly overload any reasonable system. Online algorithms are therefore not only more interesting in computational neuroscience, but also for technical applications.

### 6.3.4 Self-organizing network architectures and genetic algorithms

In the previous discussions we have always started with a particular architecture of feedforward networks by specifying the number of layers and the number of nodes per layer. The architecture is often crucial for the abilities of such networks. The universal-approximation theorem only tells us that mapping networks with enough hidden nodes can represent any mapping function, but this does not give us any hint as to how many nodes we need in practice, nor how these nodes should be connected. Too few nodes might prevent the mapping network from being able to represent a mapping function, and too many nodes in the design can drastically diminish the generalization abilities of the network. Several algorithms have been proposed to help with the design of networks. We call these types of learning procedures design algorithms.

There is often a critical dimension with regard to the number of hidden nodes necessary for a mapping network to be able to represent a particular function to a desired accuracy. An example is shown in Fig. 6.15 where the average error (normalized sum over all training examples) and the worst error (largest error of one training example) are shown. The network was trained on the task of adding two binary 3-digit numbers, and a new hidden node was created after some fixed amount of training with the specific number of hidden nodes of a mapping network, as indicated by the numbers between the dashed lines in the figure. Although the average error decreased steadily (which can be expected as the number of parameters increases), some examples were always not well represented until seven hidden nodes were reached. This is an example of a procedure called a node creation algorithm, which uses this effect by starting with a small network and increasing the number of nodes systematically until the required performance of the network is reached. Other procedures, called pruning algorithms, start with a large number and decrease the number of nodes until a satisfactory solution is found. A particularly interesting version of a pruning algorithm is weight decay, as already discussed in Chapter 4.

Some design algorithms, which are very relevant in biological systems, are termed genetic algorithms because of their attempt to simulate evolutionary processes. Evolutionary computing and genetic algorithms are active research areas where further advances can be expected; in particular, in the area of co-evolution. The genetic algorithms discussed here are certainly vastly abstracted
from evolutionary processes in nature, but they are intended to illustrate some important basic ingredients.

Genetic algorithms work commonly on large vectors, typically called **genomes**, which describe structures to be optimized. To apply such algorithms to the design of neural networks we can, for example, define a large vector with components that are either one or zero. The ones indicate the presence, and the zeros the absence of a connection between corresponding nodes in a network. Such a connectivity vector therefore describes a particular architecture, which is the analogue to an **individual** in a biological system. The principle behind genetic algorithms is to generate a large set of such genome vectors (individuals), called a **population**, and to evaluate each single vector with the help of an **objective function**, which specifies the performance of each individual. From this population we generate a new population using **genetic operators**. There are many possible choices of such operators of which we only mention some that seem particularly important. At first there should be a **survival operator** that saves only the individuals that perform well while poor performers are discarded. A new population of new individuals (**offspring**) are generated only by the good performers. Offspring can, for example, be generated by combining parts of the genome of two good performers with an operation that is called **cross-over**. In addition, it is essential to allow some random modifications to the result. We can think of this operation as **random mutation**.

The generation of a new population with individuals that are likely to perform better than previous generations in terms of the objective function is essential for the optimization capability of a genetic algorithm. In this way genomes are optimized to perform a certain task. The application of these simple algorithms shows that many generations are necessary in order to generate satisfactory individuals. This fact often prevents the use of these algorithms for the design of networks in engineering applications. The slow convergence of such simplified algorithms also suggests that additional organizing principles that enable a more efficient optimization within evolutionary strategies might be important. It is, however, widely accepted that the principles behind evolutionary algorithms have helped to develop the major structure of the central nervous system. It is important to realize that the global structures of the brain are similar in different individuals and are therefore likely to be coded genetically. The comparison of different species has shown that such structures have been evolving over time, with evolutionary forces driving new developments. In contrast to the evolutionary forces that allowed the evolution of important structures of the brain, the learning algorithms discussed throughout the book are primarily thought to fine-tune the brain within the genetically dedicated designs. Understanding the learning abilities of neural networks within the constraints of the brain organization is therefore a major concern of computational neuroscience and should not be forgotten in the discussions of neuronal networks.

### 6.3.5 Mapping networks with context units

The study of cognitive abilities of humans reveals that our behaviour, for example the execution of particular motor actions, often depends on the context
in which we encounter a certain situation. For example, we might encounter an equivalent situation on two consecutive days, such as seeing a person in front of a house who is apparently studying the building in some detail. On the first day we might just think that this person is interested in architecture and we will likely proceed without acting further on this encounter. In the morning of the second day we might read in the newspaper about an increase in burglaries in the area, and seeing the person of the previous day again studying a house might very well prompt us to enquire about his or her intentions.

A simple architecture that demonstrates some form of contextual processing was proposed by Jeffrey Elman and is sometimes called an Elman-net. An example is outlined in Fig. 6.16, which illustrates a mapping network with four input nodes, three hidden nodes, and four output nodes. In addition to the standard feedforward mapping components, the network has context nodes. The three context nodes shown in the figure contain the activation of the hidden nodes of the previous time step. Furthermore, the activations of the context units are fed back into the system as internal inputs (rather than external inputs) for the next time step. The architecture therefore includes a new class of projections that feed into cells which, in turn, can influence the sending node at a later time. The network is therefore said to be recurrent. This type of physical back-projection should not be confused with the information flow that is used during training the networks, such as in the error-back-propagation algorithm discussed above.

The network in Fig. 6.16 is only one example of this class of networks, where we have included context nodes that receive inputs from the hidden nodes. We can also include context units that receive inputs directly from output or input nodes. The latter remember the input of the previous time step, and such a mechanism is often termed short-term memory. We will discuss related but distinct forms of short-term memory as used in the physiology and psychology literature in Chapter 8.

The context units in the example of Fig. 6.16 are designed to contain the activity of the hidden nodes at the previous time step. This can formally be achieved with linear units and setting the weight values to one while assuming some delay in the projections. The network functions as follows. For each input to the network, consisting of the external input from the input nodes and from the context nodes (which memorized the previous firing rates of the hidden nodes), we calculate the activation of the hidden nodes and then the activation of the output nodes. The activation of the hidden nodes is also copied to the context nodes. All this can be thought of as a basic time step of the network. Thus, we can treat the network at each time step as a standard feedforward mapping network and can thus use the back-propagation algorithm discussed before. However, the function of the network now has inherent time dynamics that there is a new input to the hidden nodes at the next time step (even with the same external input from the input nodes as in the previous time step). To take into account the context during training we have to train the network on whole sequences of inputs. The advantage is that the network can generate sequences of outputs, and we can thus use those models in a wider context. In particular, these networks can learn to predict the next output in sequence of symbols, and they have been used primarily in this context.
6.3.6 Probabilistic mapping networks

The outputs of a mapping network can also be interpreted as probabilities that a specific input vector has certain features represented by the individual output nodes. Such networks are particularly useful for data classification. For such applications we can employ a feedforward mapping network that has \( n_{\text{out}} \) output nodes, where \( n_{\text{out}} \) is equal to the number of possible classes to which an object represented by the input vector can belong. The activity of each output node can be interpreted as the probability of membership of the object to the class represented by the node. Such mapping networks are sometime called \textit{probabilistic feedforward networks}. Note that such probabilistic networks are different from \textit{stochastic networks} in which the updating rule of the nodes has some probabilistic (stochastic) components (discussed further in Chapter 8).

To allow a probabilistic interpretation of the activities of the output nodes one needs to normalize the sum of all output activities to one (see Appendix C),

\[
\sum_i r_i^{\text{out}} = 1. \tag{6.40}
\]

The reason for this is that the probability of the input vector belonging to any of the classes must be one if the nodes represent all possible classes. Such a condition can be achieved, for example, with an output layer that competes for the output, which can be realized with collateral connections between the output nodes as shown in Fig. 6.17. These collateral connections can be inhibitory so that the strong activity of one node inhibits the firing of other nodes. In the extreme case of very strong inhibition, so that only one node is active for each input vector, we speak of a \textit{winner-take-all} architecture. Such competitive networks with collateral connections are discussed in Chapter 7. Here we only want to stress the probabilistic interpretation of the output of mapping networks.

A winner-take-all output layer in a mapping network can only indicate the class to which an object represented by an input vector is most likely to belong. More useful is a representation of class-membership probability, so that we can also gain some information on the confidence with which the network has classified the input vector. This can be achieved with soft competition in the output layer. In practice we can simulate this with a normalization of the output activity using the \textit{softmax} function

\[
r_i^{\text{out}} = \frac{e_i^{\text{out}}}{\sum_j e_j^{\text{out}}}, \tag{6.41}
\]

where \( r_i^{\text{out}} \) are the firing rates of the output nodes before the normalization step, and \( r_i^{\text{out}} \) is the final output of the output nodes that can be interpreted as probabilities of class membership because \( \sum_j r_j^{\text{out}} = 1 \). This can be implemented with a feedforward network as illustrated in Fig. 6.18, where the new output layer has a (non-local) softmax activation function.

Many learning algorithms are based on performance measures that represent the objective function. For the probabilistic interpretation it is appropriate to
6.4 Support vector machines

In this chapter we walked a fine line between biological plausibility and solving a general statistical learning problem. Perceptrons have shown us that feedforward networks of neuron-like (or population) elements have interesting capabilities and that learning in such systems, at least without hidden units, has signatures of Hebbian plasticity. More formally, we have seen that MLPs are universal approximators in the sense that correct mappings between training data can be learned given enough hidden nodes. However, there remain many problems with such machines, specifically the efficiency of learning and their generalization abilities. We encountered the inefficiency of learning by many training cycles when training an MLP on the XOR example, and larger problems often suffer from slow convergence and local minima. Also, it is not only the learning (memorization) of training data that are the important part of applications of such machines. The real challenge comes when applying such machines to unseen data. MLPs are known to be fairly good interpolators, but extrapolation of data is often weak. We have seen some pure generalization ability in the letter example in that the trained networks were not very robust against noise.

Many problems of MLPs have now been solved by advanced machine learning methods. In this final section, we briefly review machine learning in some more generality and introduce the basic ideas behind support vector machines which have now replaced MLPs in many applications. While this is a brief deviation of discussing brain processing, this discussion is important for two reasons. First, machine learning had historically a close connection to brain research, and many advancements of theoretical descriptions of brain processing have been derived in this scientific discipline. Second, machine learning methods are becoming essential tools in neuroscience for data analysis. For example, machine learning tools have been essential in analysing brain imaging data to build brain–computer interfaces. The material covered in this section was largely developed by the Russian scientist Vladimir Vapnik.

6.4.1 Large-margin classifiers

Machine learning problems can be classified into regression and classification problems. In regression problems one tries to find suitable interpolations of
data points, whereas in classification problems one tries to give data a label of
different categories. Here we discuss primarily classification problems, although
the methods mentioned here can be generalized to regression problems. We
speak about binary classification if there are only two categories. Learning
Boolean functions is an example of a binary classification problem. A more
general binary classification problem is illustrated in Fig. 6.19. A solution that
can be found by a perceptron is included as the dashed line. This line is likely
to be close to some points as learning ceases after all training examples are
correctly classified.

Two interesting points can be made with this example. The first point is
that most of the data are not essential for classification. Only data points close
to the decision boundary are essential where the additional data points do not
contribute information to the best decision boundary. The second point is that
it is likely that some future data are misclassified with the perceptron solution
since the training examples might not have included all the points close to this
line or if noise is present in the measurements. A better solution, indeed the
best solution in this situation, is to choose a line (decision boundary) that is
furthest from all points. Such a solution is indicated as the shaded area. The
best solution is the decision surface in the middle of this area when the width
of the area is maximized. The distance from the middle line to the border is
called the margin. Large-margin classifiers are thus typically more robust than
perceptrons.

It is instructive to formalize this observation somewhat. We have already
seen (formula in Fig. 6.5A) that a line in this graph can be written as

\[ w_1 x_1 + w_2 x_2 - \theta = 0. \]  

(6.43)

The lines on the edges of the shaded areas are similarly

\[ w_1 x_1 + w_2 x_2 - \theta = 1 \]  

(6.44)

\[ w_1 x_1 + w_2 x_2 - \theta = -1, \]  

(6.45)

where we used a freedom of scale by setting the offsets of the lines to the
separating line (eqn 6.43) to one. The distance of these lines to the origin
can be calculated from basic trigonometry and is given by \((\theta + 1)/|w|\) and
\((\theta - 1)/|w|\), respectively. where \(|w| = \sqrt{w_1^2 + w_2^2}\) is the norm of the vector \(w\),
and the distance between the lines is the size of the margin,

\[ d = \frac{2}{|w|}. \]  

(6.46)

Maximizing the margin is therefore equivalent to minimizing the weights, which
was already a common trick for perceptrons, as mentioned in Section 6.2.2. Of
course, the maximization of the margin has to be subject to the restrictions
that there are no data points in the margin. That is, all data points of a
class to which we give the label \(y = +1\) should be above the line defined
by eqn 6.44, and the data points of the other class, to which we give label
\(y = -1\), should be below the line defined by eqn 6.45. These conditions can be
written as inequalities in a compact way. Also, we can generalize the equations
to arbitrary dimensions by using vectors for the weights (row vectors in our

---

\(^1\)This linear separable classification problem is a generalization of the
Boolean OR function illustrated in Fig. 6.5A.

\(^2\)We can set the offset to an arbitrary value, \(m\), and then divide the whole
equation by this number. Eqns 6.44
and 6.45 are then recovered by chang-
ing notations with the mappings \(w_1 \leftarrow w_1/m\); \(w_2 \leftarrow w_2/m\) and \(\theta \leftarrow \theta/m\).
standard notation) and data points (column vectors), \( w_1x_1 + w_2x_2 + \ldots = wx \). In addition, with the choice of class labels, we can multiply these equations by \( y \) so that the two eqns 6.44 and 6.45 can be summarized into one,

\[
y(wx - \theta) - 1 \leq 0. \tag{6.47}
\]

The optimization problem, minimizing the weights, or maximizing \( 1/2|w|^2 \), subject to the constraints that no points are in the margin, eqn 6.47, can be solved by using the Lagrange formalism. For this we define a Lagrange function, \( L_P \), which is a form of a cost function in which the constraints are added with so-called Lagrange multipliers, \( \alpha_i \), for each training example \( i \),

\[
L_P = \frac{1}{2}|w|^2 + \sum_i \alpha_i y_i (wx_i - \theta) + \sum_i \alpha_i. \tag{6.48}
\]

Minimizing \( L_P \) is a quadratic optimization problem which can easily and efficiently be solved with standard numerical methods. Also, minimizing the so-called primal Lagrangian, \( L_P \), is equivalent to maximizing the dual problem defined by the dual Lagrangian

\[
L_D = \sum_i \alpha_i - \frac{1}{2} \sum_{ij} \alpha_i \alpha_j y_i y_j x_i x_j. \tag{6.49}
\]

We will come back to this formula shortly. Note that only data points on the margins are relevant at the end, which are the only data points for which the corresponding Lagrange multipliers, \( \alpha \), are not zero. These data points are called support vectors, and the classifiers discussed in this section are called linear support vector machines (SVMs).

### 6.4.2 Soft-margin classifiers and the kernel trick

So far, we have only discussed linear separable problems in this section. The real challenge usually starts when considering non-linear problems and situations with overlapping data. Let's first tackle non-linear problems, such as the example plotted in Fig. 6.20A. There we plotted data of two classes on a line, which is hence a one-dimensional data space. These data can not be separated linearly. However, we can solve the problem by transforming the data with a vector function \( \phi(x) \), by including higher moments or combinations of the original feature values in a new data vector. For example, let us consider here the simple example

\[
x \rightarrow \tilde{x} = \phi(x) = \begin{pmatrix} x \\ x^2 \end{pmatrix}. \tag{6.50}
\]

The transformed data, \( \tilde{x} \), are then lying on a parabola, as shown in Fig. 6.20B, and these data can be separated linearly. Note that the data are still on a line

![Fig. 6.20 (A) Illustration of linear not separable classification problem that can be solved by transforming the data as illustrated in (B).](image)
while we consider now a data space in two dimensions. The transformation does therefore appear as increasing the dimensionality of the problem.

The challenge is now to find appropriate functions $\phi(x)$. However, we note that a lot of functions would work in this example, and the precise form of $\phi(x)$ might therefore not be crucial. Furthermore, by inserting the transformed data into eqn 6.49 we see that the function only appears in the inner product of the data, $\phi(x_i)\phi(x_j)$. We can replace this with a kernel function of the inner product of the original data,

$$\phi(x_i)\phi(x_j) = K(x_i, x_j).$$  \hspace{1cm} (6.51)

This has many practical advantages. For example, even if the transformation $\phi(x)$ transforms the data space formally into an infinite dimensional space, it is still possible that the kernel function provides finite values. Furthermore, with the right choice of kernel functions we still end up with a convex optimization problem that has no local minima and can be efficiently solved numerically. A common choice that fulfils these conditions is the Gaussian kernel

$$K(x_i, x_j) = e^{-\frac{(x_i - x_j)^2}{2\sigma^2}}.$$  \hspace{1cm} (6.52)

While this kernel introduces a new parameter, $\sigma$, it has been shown through many applications that this kernel can solve many non-linear problems. Kernel machines have been the subject of much interest in the machine learning community. While further motivations of using kernels can be made, using kernels is strictly speaking only a trick; albeit a highly successful.

How about the problem of overfitting as mentioned for regression problems in Section 6.2.2? When solving non-linear problems with the kernel SVMs we face again a decision about distinguishing complicated boundaries from overlapping data as illustrated in Fig. 6.21. Are these data truly separated by the solid line, or do the data have a simpler decision boundary, such as the dotted line, and are only convoluted by noise? Solving the problem requires additional knowledge which is usually not present at this level of data analysis. A common approach is to assume some smoothness of the problem by including some regularization constraints. Such a constraint is included in SVMs by minimizing the weights, and it has been argued that this works well with kernel functions and implements a form of structural risk minimization. However, in the non-linear case, we need to allow misclassification of training data, which is done by softening the classification conditions of eqn 6.47 by allowing misclassifications with a penalty value determined by a new parameter $C$. This is an additional parameter that must be tuned in the standard SVM machines. Vapnik has greatly contributed to the theory of structural risk minimization that we will not discuss further. The list in 'Further reading' includes some references to good tutorials and books where more discussions can be found.

Simulation

We show here two simulations using the SVM toolbox called LIBSVM. These library functions, which provide interfaces for many programming languages including MATLAB, were written by Chih-Chung Chang and Chih-Jen Lin.
and can be downloaded from http://www.csie.ntu.edu.tw/~cjlin/libsvm. We use here the default parameters of these packages, which include using a Gaussian kernel function.

The first example, shown in Table 6.11 is training the SVM on the Boolean XOR function. Training is provided in function `svmtrain` that takes a column vector of labels and a matrix of data points as inputs. The trained model is passed by the data structure `model` to the prediction routine. This function also takes labels as input, but these labels are only used to evaluate the performance. Running the program returns the line

Accuracy = 100% (4/4) (classification)

which demonstrates that the function was correctly learned by the SVM.

The second example, shown in Table 6.12, consists of overlapping data. We choose data from one class randomly with an equal distribution in a square, and draw data of the second class from a Gaussian distribution centred in the middle of the square. It is not possible to classify the data perfectly, but it is more likely that data close to the centre of the square are from the second class. The 100 data points for each class used to train the SVM are shown with different symbols for each class in Fig. 6.22. We tested the trained SVM on a grid of data to evaluate which areas are labelled by the SVM with different
classes. We used thereby a vector with only one label in the evaluation function as this was not needed for these experiments. The accuracy reported by the function does not, therefore, reflect the true accuracy of the SVM. The area in which the SVM predicts data points from the second class is shown as the grey area in Fig. 6.22. When evaluating the performance over 100 trials one gets, for example, an accuracy of 87.2 ± 2.2% which is very good, considering that the maximal achievable accuracy is 88.2% for this example. The last value can be calculated from the integrals over the density functions in the areas where each class is dominating.

Exercises

(6.1) Implement a single-layer perceptron and train it to translate the digital letters given in file pattern1 into the corresponding ASCII representation. Plot a training curve and interpret your results.

(6.2) Implement an MLP and train it to translate the digital letters given in file pattern1 into the corresponding ASCII representation. Plot
Further reading

There are many books on artificial neural networks with emphasis on different aspects. One of the best books on the engineering aspects of neural networks is the book by Simon Haykin (1999), and the books by Hertz, Krogh, and Palmer (1991) and Müller, Reinhardt, and Strickland (1995) are some great introductions. A modern view of neural networks and machine learning theory is provided by Christopher Bishop (2006). A good tutorial on SVMs, which includes discussions of structural risk minimization and the VC dimension, is given by Burges (1998), and a good tutorial on SVM regression by people who have greatly contributed to this area is given by Smola and Schölkopf (2004). The lectures by Andrew Ng (2008) on machine learning are highly recommended for further studies of this area.

Neural networks had a profound influence on cognitive science in the form of connectionist models since the classic PDP books (Rumelhart et al., 1986). Another example of a book that uses connectionist modelling to understand cognitive processes is McLeod et al. (1998). This book includes a discussion of Elman networks, which were mentioned only briefly in this chapter. This chapter only scratched the surface of perception and sensation. A thorough introduction to this area, including the psychophysics and the underlying physiology, can be found in this book by Goldstein (1999).


David E. Rumelhart, James L. McClelland, and the PDP research group (1986), Parallel distributed processing: Explorations in the microstructure of cognition, MIT Press.


a training curve and interpret your results.

(6.3) Investigate how much noise the perceptrons can tolerate in the pattern before being unable to recognize a letter.

(6.4) Which letter is represented in file pattern2?

(6.5) Calculate the maximal achievable accuracy in the overlapping data example of the last simulation connection.