Rote Learning

The simplest way for a computer to learn from experience is simply to learn by **rote**. Training involves storing each piece of training data and its classification. Thereafter, a new item of data is classified by looking to see if it is stored in memory. If it is, then the classification that was stored with that item is returned. Otherwise, the method fails.

Learning Concepts

We will now look at a number of methods that can be used to learn concepts. **Concept learning** involves determining a mapping from a set of input variables to a Boolean value.

The methods described here are known as **inductive-learning methods**. These methods are based on the principle that if a function is found that correctly maps a large set of training data to classifications, then it will also correctly map unseen data. In doing so, a learner is able to **generalize** from a set of training data.

To illustrate these methods, we will use a simple toy problem, as follows:

Our learning task will be to determine whether driving in a particular manner in particular road conditions is safe or not. We will use the following attributes:

Attribute	Possible values
Speed	slow, medium, fast
Weather	wind, rain, snow, sun
Distance from car in front	10ft, 20ft, 30ft, 40ft, 50ft, 60ft
Units of alcohol driver has drunk	0, 1, 2, 3, 4, 5
Time of day	morning, afternoon, evening, night
Temperature	cold, warm, hot

We will consider a **hypothesis** to be a vector of values for these attributes. A possible hypothesis is

h_1 = <slow, wind, 30ft, 0, evening, cold>

We also want to represent in a hypothesis that we do not care what value an attribute takes. This is represented by "?", as in the following hypothesis:

h_2 = <fast, rain, 10ft, 2, ?, ?>

h_2 represents the hypothesis that driving quickly in rainy weather, close to the car in front after having drunk two units of alcohol is safe, regardless of the time of day or the temperature. Clearly, this hypothesis is untrue and would be considered by the learner to be a **negative training example**.

In other cases, we need to represent a hypothesis that no value of a particular attribute will provide a positive example. We write this as "∅", as in the following hypothesis:

h_3 = <fast, rain, 10ft, 2, ∅, ∅>

h_3 states the opposite of h_2—that driving quickly in rainy weather, close to the car in front after having drunk two units of alcohol cannot be safe, regardless of the time of day or the temperature.

The task of the concept learner is to examine a set of positive and negative training data and to use these to determine a hypothesis that matches all the training data, and which can then be used to classify instances that have not previously been seen.

Concept learning can be thought of as search through a search space that consists of all possible hypotheses, where the goal is the hypothesis that most closely represents the correct mapping.

General-to-Specific Ordering

Consider the following two hypotheses:

h_g = <?, ?, ?, ?, ?, ?>

h_s = <∅,∅,∅,∅,∅,∅>

h_g is the hypothesis that it is safe to drive regardless of the conditions—this is the **most general hypothesis**.

h_s is the **most specific hypothesis**, which states that it is never safe to drive, under any circumstances.

These hypotheses represent two extremes, and clearly a useful hypothesis that accurately represents the mapping from attribute values to a Boolean value will be somewhere in between these two.

One method for concept learning is based on the idea that a **partial order** exists over the space of hypotheses. This partial order is represented by the relationship "more general than":

$\geq_g$

We write

$h_1 \geq_g h_2$

which states that h_1 is more general than (or as general as) h_2. Similarly, we could write

$h_1 >_g h_2$

in the case where h_1 is certainly more general than h_2.

$\geq_g$ defines a partial order over the hypothesis space, rather than a total order, because some hypotheses are neither more specific nor more general than other hypotheses. For example, consider the following hypotheses:

h_1 = <?, ?, ?, ?, evening, cold>

h_2 = <medium, snow, ?, ?, ?, ?>

We cannot express any relationship between h_1 and h_2 in terms of generality.

One hypothesis is more general than (or equally general as) another hypothesis if every instance that is matched by the second hypothesis is also matched by the first hypothesis.

For example,

<slow, ?, ?, ?, ?, ?> $\geq_g$ <slow, ?, ?, ?, ?, cold>

It should be clear that a more general hypothesis matches more instances than a less general hypothesis.

A Simple Learning Algorithm

The following algorithm uses the general-to-specific ordering of hypotheses to search the hypothesis space for a suitable hypothesis. The method is as follows:

Start with the most specific hypothesis. In our example above, this would be <∅, ∅, ∅, ∅, ∅, ∅, ∅, ∅>.

Now, for each positive training example, determine whether each attribute in the example is matched by the current hypothesis. If it is not, replace the attributes in the hypothesis with the next more general value that does match.

For example, let us consider the following set of positive training data:

<slow, wind, 30ft, 0, evening, cold>

<slow, rain, 20ft, 0, evening, warm>

<slow, snow, 30ft, 0, afternoon, cold>

First, let us compare the first item of training data with the current hypothesis, which is <∅, ∅, ∅, ∅, ∅, ∅, ∅, ∅>. Clearly, none of the attributes are matched by this hypothesis. The next most general value for each attribute than ∅ that matches the training data is the value contained in the training data. So, we replace our hypothesis with the following hypothesis:

<slow, wind, 30ft, 0, evening, cold>

Clearly, the hypothesis <?, ?, ?, ?, ?, ?, ?, ?> would have been more general than the initial hypothesis, but the method we are using is to select the *next more general* value for each attribute. In this way, we move from a hypothesis that is too specific to one that is general enough to match all the training data.

We will now consider the second item of training data:

<slow, rain, 20ft, 0, evening, warm>

Now we compare each attribute value with the corresponding value in our current hypothesis. Where the values match, we do not need to make any change. Where they do not match, we need to replace the value with "?" so that the hypothesis matches both items of training data. Hence, our new hypothesis is

<slow, ?, ?, 0, evening, ?>

By comparing with our final item of training data, we arrive at the following hypothesis:

<slow, ?, ?, 0, ?, ?>

This hypothesis states that it is only safe to drive if one drives slowly and has not drunk any alcohol, and that this is true regardless of the road or weather conditions.

This hypothesis is **consistent** with the training examples, which means that it maps each of them to the correct classification.

This algorithm will generate the most specific hypothesis that matches all of the training data. There are a number of problems with this algorithm: first of all, it may not be desirable to identify the most specific hypothesis—it may be that the most general hypothesis that matches the training data provides a better solution. Secondly, the most specific hypothesis identified by the algorithm may not be the only solution—there may be other most specific hypotheses that match the data, one of which may be a preferable solution. Additionally, this algorithm does not make any use of negative examples. As we will see, most useful learning methods are able to make use of negative as well as positive training examples.

Finally, the method does not deal well with inconsistent or incorrect training data. In real-world problems, an ability to deal with such errors is vital, as we see later in this part of the book.

Version Spaces

Given a set of training examples (positive and negative), the set of hypotheses that correctly map each of the training examples to its classification is called the **version space**.

One method for learning from a set of data is thus to start from a complete version space that contains all hypotheses and systematically remove all the hypotheses that do not correctly classify each training example. Although this method might work well on small problems, for problems of any reasonable size, the task of enumerating all hypotheses would be impractical.

Candidate Elimination

We now explore another method that uses version spaces to learn. The aim of these methods is to identify a single hypothesis, if possible, that correctly describes the problem. The more training data that are available, the fewer hypotheses are contained in the version space. If all the training data have been used, and the version space contains just a single hypothesis, then this matches all the training data and should also match unseen data.

The **candidate elimination** learning method operates in a similar manner to the simple algorithm presented in Section 10.5.1. Unlike the earlier simple method, the candidate elimination method stores not just a single hypothesis, but two sets of hypotheses. In addition to maintaining a set of most specific hypotheses that match the training data, this method also maintains a set of hypotheses that starts out as a set with the single item

<?, ?, ?, ?, ?, ?, ?, ?> and ends up being a set of the most general hypotheses that match all the training data. This algorithm is thus able to make use of negative training data as well as positive training data.

The method operates as follows: Two sets are maintained of hypotheses, h_s and h_g:

h_s is initialized as {<∅, ∅, ∅, ∅, ∅, ∅, ∅, ∅>} and

h_g is initialized as {<?, ?, ?, ?, ?, ?, ?, ?>}.

When a positive training example is encountered, it is compared with the hypotheses contained in h_g. If any of these hypotheses does not match the training example, it is removed from h_g. The positive training data are then compared with the hypotheses contained in h_s. If one of these hypotheses does not match the training data, it is replaced by the set of slightly more general hypotheses that are consistent with the data, and such that there is at least one hypothesis in h_g that is more general.

This method is applied in reverse for negative training data. By applying this method to each item of training data, the sets h_g and h_s move closer to each other and eventually between them contain the full version space of hypotheses that match all the training data.

Inductive Bias

All learning methods have an *inductive bias*. Inductive bias refers to the restrictions that are imposed by the assumptions made in the learning method. For example, in the above discussions we have been assuming that the solution to the problem of road safety can be expressed as a con- junction of a set of eight concepts. This does not allow for more complex expressions that cannot be expressed as a conjunction. This inductive bias means that there are some potential solutions that we cannot explore, and which are, therefore, not contained within the version space we examine.

This may seem like an unfortunate limitation, but in fact inductive bias is essential for learning. In order to have an unbiased learner, the version space would have to contain every possible hypothesis that could possibly be expressed. This would impose a severe limitation: the solution that the learner produced could never be any more general than the complete set of training data. In other words, it would be able to classify data that it had previously seen (as the rote learner could) but would be unable to generalize in order to classify new, unseen data.

The inductive bias of the candidate elimination algorithm is that it is only able to classify a new piece of data if all the hypotheses contained within its version space give the data the same classification. Hence, the inductive bias does impose a limitation on the learning method.

In the 14th century, William of Occam proposed his famous "**Occam's razor**," which simply states that it is best to choose the simplest hypothesis to explain any phenomenon. We can consider this to be a form of inductive bias, which states that the best hypothesis to fit a set of training data is the simplest hypothesis. We will see later how this inductive bias can be useful in learning decision trees.

Decision-Tree Induction

In Chapter 3, we see a tree that was used to determine which species a particular bird belonged to, based on various observed features of the bird. A variation of this kind of tree, where the leaf nodes are all Boolean values is called a **decision tree**. A decision tree takes in a set of attribute values and outputs a Boolean decision.

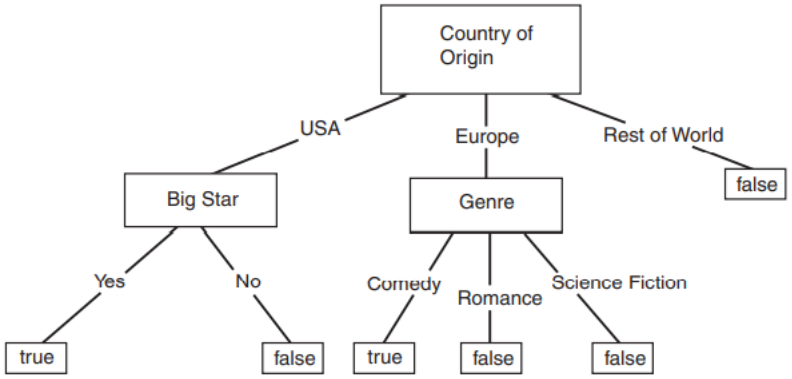

Figure 10.1
A simple decision tree for determining whether or not a film will be a box-office success

An example of a decision tree is shown in Figure 10.1. This decision tree can be used to determine whether or not a given film will be a success at the box office.

To use the decision tree, we start at the top and apply the question to the film. If the film is made in the United States, we move down the first branch of the tree; if it is made in Europe the second; and if elsewhere then we explore the third branch. The final boxes represent the Boolean value, true or false, which expresses whether a film is a success or not.

According to this extremely simplistic (and possibly somewhat contentious) decision tree, a film can only be a box-office success if it is made in the United States and has a big star, or if it is a European comedy.

Whereas version spaces are able to represent expressions that consist solely of conjunctions, decision trees can represent more complex expressions, involving disjunctions and conjunctions. For example, the decision tree in Figure 10.1 represents the following expression:

$$((\text{Country} = \text{USA}) \wedge (\text{Big Star} = \text{yes})) \vee ((\text{Country} = \text{Europe}) \wedge (\text{Genre} = \text{comedy}))$$

Decision-tree induction (or decision-tree learning) involves using a set of training data to generate a decision tree that correctly classifies the training data. If the learning has worked, this decision tree will then correctly classify new input data as well.

The best-known decision tree induction algorithm is ID3, which was developed by Quinlan in the 1980s.

The ID3 algorithm builds a decision tree from the top down. The nodes are selected by choosing features of the training data set that provide the most information about the data and turning those features into questions. For example, in the above example, the first feature to be noted might be that the country of origin is a significant determinant of whether a film will be a success or not. Hence, the first question to be placed into the decision tree is “what is the film’s country of origin?”.

The most important feature of ID3 is how the features are chosen. It would be possible to produce a decision tree by selecting the features in an arbitrary order, but this would not necessarily produce the most efficient decision tree. The ID3 algorithm finds the shortest possible decision tree that correctly classifies the training data.

Information Gain

The method used by ID3 to determine which features to use at each stage of the decision tree is to select, at each stage, the feature that provides the greatest **information gain**. Information gain is defined as the reduction in entropy. The entropy of a set of training data, S, is defined as

$$H(S) = -p_1 \log_2 p_1 - p_0 \log_2 p_0$$

where p_1 is defined as the proportion of the training data that includes positive examples, and p_0 is defined as the proportion that includes negative examples. The entropy of *S* is zero when all the examples are positive, or when all the examples are negative. The entropy reaches its maximum value of 1 when exactly half of the examples are positive and half are negative.

The information gain of a particular feature tells us how closely that feature represents the entire target function, and so at each stage, the feature that gives the highest information gain is chosen to turn into a question.

Example

We will start with the training data given below:

Film	Country of origin	Big star	Genre	Success
Film 1	United States	yes	Science Fiction	true
Film 2	United States	no	Comedy	false
Film 3	United States	yes	Comedy	true
Film 4	Europe	no	Comedy	true
Film 5	Europe	yes	Science fiction	false
Film 6	Europe	yes	Romance	false
Film 7	Rest of World	yes	Comedy	false
Film 8	Rest of World	no	Science fiction	false
Film 9	Europe	yes	Comedy	true
Film 10	United States	yes	Comedy	true

We will now calculate the information gain for the three different attributes of the films, to select which one to use at the top of the tree.

First, let us calculate the information gain of the attribute “country of origin.” Our collection of training data consists of five positive examples and five negative examples, so currently it has an entropy value of 1.

Four of the training data are from the United States, four from Europe, and the remaining two from the rest of the world.

The information gain of this attribute is the reduction in entropy that it brings to the data. This can be calculated as follows:

First, we calculate the entropy of each subset of the training data as broken up by this attribute. In other words, we calculate the entropy of the items that are from the United States, the entropy of the items from Europe, and the entropy of the items from the rest of the world.

Of the films from the United States, three were successes and one was not. Hence, the entropy of this attribute is

$$H(USA) = - (3/4) \log2 (3/4) - (1/4) \log2 (1/4)$$
$$= 0.311 + 0.5$$
$$= 0.811$$

Similarly, we calculate the entropies of the other two subsets as divided by this attribute:

$$H(Europe) = 1$$

(since half of the European films were successes, and half were not).

$$H(Rest\ of\ world) = 0$$

(since none of these films were successes).

The total information gain is now defined as the original entropy of the set minus the weighted sum of these entropies, where the weight applied to each entropy value is the proportion of the training data that fell into that category. For example, four-tenths of the training data were from the United States, so the weight applied to H(USA) is 4/10 = 0.4.

The information gain is defined as:

$$Gain = 1 — (0.4 \times 0.811) — (0.4 \times 1) — (0.2 \times 0)$$
$$= 1 — 0.3244 — 0.4 — 0$$
$$= 0.2756$$

Hence, at this stage, the information gain for the "country of origin" attribute is 0.2756.

For the "Big star" attribute

$$H(yes) = 0.9852$$
$$H(no) = 1$$

so, the information gain for this attribute is

$$Gain = 1 — (0.7 \times 0.9852) — (0.3 \times 1)$$
$$= 1 — 0.68964 — 0.3$$
$$= 0.01$$

For the "Genre" attribute

$$H(science\ fiction) = 0.918296$$
$$H(comedy) = 0.918296$$
$$H(romance) = 0$$

(note that we treat $0 \times \log_2 0$ as 0)

hence, the information gain for this attribute is

$$Gain = 1 — (0.3 \times 0.918296) — (0.6 \times 0.918296) — (0.1 \times 0)$$
$$= 1 — 0.2754888 — 0.5509776 — 0$$
$$= 0.17$$

Hence, at this stage, the category "Country of origin" provides the greatest entropy gain and so is placed at the top of the decision tree. This method is then applied recursively to the sub-branches of the tree, examining the entropy gain achieved by subdividing the training data further.

Inductive Bias of ID3

ID3's inductive bias is that it tends to produce the shortest decision tree that will correctly classify all of the training data. This fits very well with Occam's razor, which was briefly introduced in Section 10.8. It is not the case that Occam's razor can be applied in all situations to provide the optimal solution: it is, however, the case that ID3 tends to produce adequate results. Additionally, a smaller decision tree is clearly easier for humans to understand, which in some circumstances can be very useful, for example if the need arises to debug the learner and find out why it makes a mistake on a particular piece of unseen data.

The Problem of Overfitting

In some situations, decision trees (and other learning methods) can run into the problem of overfitting. Overfitting usually occurs when there is noise in the training data, or when the training data do not adequately rep- resent the entire space of possible data. In such situations, it can be possible for one decision tree to correctly classify all the training data, but to per- form less well at classifying unseen data than some other decision tree that performs poorly at classifying the training data. In other words, if the training data do not adequately and accurately represent the entire data set, the decision tree that is learned from it may not match unseen data.

This problem does not just apply to decision trees, but also to other learning methods. It can best be understood by examining the illustration in Figure 10.2.

In the first diagram in Figure 10.2, black dots are positive training data, and white dots are negative training data. The two lines represent two hypotheses that have been developed to distinguish the training data. The thin line is a relatively simple hypothesis, which incorrectly classifies some of the training data—it should have all positive examples below it and all negative examples above it. The thicker line correctly classifies all the training data, using a more complex hypothesis, which is somewhat warped by noise in the data. In the next diagram, the thin line is shown to map reasonably effectively the full set of data. It does make some errors, but it reasonably closely represents the trend in the data. The third diagram, however, shows that the more complex solution does not at all represent the full set of data. This hypothesis has been overfitted to the training data, allowing itself to be warped by noise in the training data.

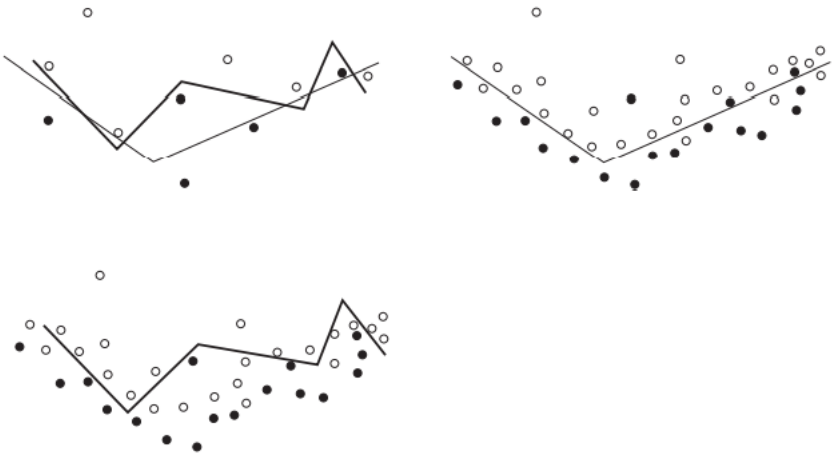

Figure 10.2
Illustration of the problem of overfitting

Overfitting is, perhaps, a good illustration of why Occam's razor can some- times be a useful inductive bias: selecting a complex solution to accommodate all of the training data can be a bad idea when the training data contain errors.

The Nearest Neighbor Algorithm

The **nearest neighbor algorithm** is an example of instance-based learning. Instance-based learning methods do not attempt to generalize from training data to produce a hypothesis to match all input data, instead, they store the training data and use these data to determine a classification for each new piece of data as it is encountered.

The nearest neighbor algorithm operates in situations where each instance can be defined by an *n*-dimensional vector, where *n* is the number of attributes used to describe each instance, and where the classifications are discrete numerical values. The training data are stored, and when a new instance is encountered it is compared with the training data to find its nearest neighbors. This is done by computing the Euclidean distance between the instances in n-dimensional space. In two-dimensional space, for example, the distance between $<x_1, y_1>$ and $<x_2, y_2>$ is

$$\sqrt{\left((x_1 - x_2)^2 + (y_1 - y_2)^2\right)}$$

Typically, the nearest neighbor algorithm obtains the classifications of the nearest *k* neighbors to the instance that is to be classified and assigns it the classification that is most commonly returned by those neighbors.

An alternative approach is to weight the contribution of each of the neighbors according to how far it is from the instance that is to be classified. In this way, it is possible to allow every instance of training data to contribute to the classification of a new instance. When used in this way, the algorithm is known as **Shepard's method**.

Unlike decision-tree learning, the nearest neighbor algorithm performs very well with noisy input data. Its inductive bias is to assume that instances that are close to each other in terms of Euclidean distance will have similar classifications. In some cases, this can be an erroneous assumption; for example, in a situation where 10 attributes are used to define each instance, but only 3 of those attributes play any part in determining the classification of the instance. In this situation, instances can be very far apart from each other in 10-dimensional space and yet have the same classification. This problem can be avoided to some extent by neglecting to include unimportant attributes from the calculations.

Learning Neural Networks

An artificial neural network is a network of simple processing nodes, which is roughly modeled on the human brain. The human brain is a massively parallel computation device, which achieves its power through the enormous connectivity between its neurons. Each neuron is a very simple device that can either fire or not fire, but by combining billions of these neurons together, the brain is able to achieve levels of complexity as yet unattainable by machines.

The word artificial is often used to describe neural networks to differentiate them from the biological neural networks that make up the human brain, but in this book we shall simply refer to them as neural networks because it should be clear from the context which type of network we are referring to.

Neural networks consist of a number of nodes, each of which can be thought of as representing a neuron. Typically, these neurons are arranged into layers, and the neurons from one layer are connected to the neurons in the two layers on either side of it.

Typically, the network is arranged such that one layer is the input layer, which receives inputs that are to be classified. These inputs cause some of the neurons in the input layer to fire, and these neurons in turn pass signals to the neurons to which they are connected, some of which also fire, and so on. In this way, a complex pattern of firings is arranged throughout the network, with the final result being that some neurons in the final output layer fire.

The connections between neurons are weighted, and by modifying these weights, the neural network can be arranged to perform extremely complex classification tasks such as handwriting analysis and face recognition.

As we see in Chapter 11 where we discuss them in more detail, neural networks have a number of advantages over other learning methods. Many of these advantages derive from features of the human brain. For example, neural networks are extremely **robust**, both to errors in any training data and to damage that may be caused to the network itself.

Supervised Learning

Supervised learning networks learn by being presented with preclassified training data. The techniques we have discussed so far in this chapter use forms of supervised learning. Neural networks that use supervised learning learn by modifying the weights of the connections within their networks to classify the training data more accurately. In this way, neural networks are able to generalize extremely accurately in many situations from a set of training data to the full set of possible inputs.

One of the most commonly used methods for supervised learning is back- propagation, which will be discussed in Chapter 11.

Unsupervised Learning

Unsupervised learning methods learn without any human intervention. A good example of an unsupervised learning network is a **Kohonen map**. A Kohonen map is a neural network that is able to learn to classify a set of input data without being told what the classifications are and without being given any training data. This method is particularly useful in situations where data need to be classified, or clustered, into a set of classifications but where the classifications are not known in advance.

For example, given a set of documents retrieved from the Internet (perhaps by an intelligent information agent), a Kohonen map could cluster similar documents together and automatically provide an indication of the distinct subjects that are covered by the documents.

Another method for unsupervised learning in neural networks was proposed by Donald Hebb in 1949 and is known as Hebbian learning. Hebbian learning is based on the idea that if two neurons in a neural network are connected together, and they fire at the same time when a particular input is given to the network, then the connection between those two neurons should be strengthened. It seems likely that something not dissimilar from Hebbian learning takes place in the human brain when learning occurs (Edelman 1987).

Reinforcement Learning

Classifier systems, which are discussed in Chapter 13, use a form of **reinforcement learning**. A system that uses reinforcement learning is given a positive reinforcement when it performs correctly and a negative reinforcement when it performs incorrectly. For example, a robotic agent might learn by reinforcement learning how to pick up an object. When it successfully picks up the object, it will receive a positive reinforcement.

The information that is provided to the learning system when it performs its task correctly does not tell it why or how it performed it correctly, simply that it did.

Some neural networks learn by reinforcement. The main difficulty with such methods is the problem of **credit assignment**. The classifier systems (which are discussed in Chapter 13) use a **bucket brigade algorithm** for deciding how to assign credit (or blame) to the individual components of the system. Similar methods are used with neural networks to determine to which neurons to give credit when the network performs correctly and which to blame when it does not.

Chapter Summary

- ✓ Many learning methods use some form of training to learn to generalize from a set of preclassified training data to be able to correctly classify unseen data.
- ✓ Rote learning involves simply memorizing the classifications of training data. A rote learning system is not able to generalize and so is only able to classify data it has seen before.
- ✓ A general-to-specific ordering of hypotheses can be used to learn to generalize from a set of training data to a hypothesis that matches all input data. This is known as concept learning.
- ✓ A version space, which consists of all possible hypotheses that match a given set of training data, can be used to generalize from those training data to learn to classify unseen data.
- ✓ Candidate elimination is a method that uses the general-to-specific ordering to produce a set of hypotheses that represent the entire version space for a problem.
- ✓ The inductive bias of a learning method is the assumptions it makes about the possible hypotheses that can be used. A learning system with no inductive bias is not capable of generalizing beyond the training data it is given.
- ✓ Decision-tree induction can be used to learn a decision tree that will correctly classify a set of input data. The inductive bias of decision-tree induction is to prefer shorter trees.
- ✓ The problem of overfitting occurs when there is noise in the training data that causes a learning method to develop a hypothesis that correctly matches the training data but does not perform well on other input data.
- ✓ The nearest neighbor algorithm simply memorizes the classifications of the training data, and when presented with a new piece of data gives the majority answer given by the closest neighbors to this piece of data in n-dimensional space.
- ✓ Neural networks are based on biological networks of neurons contained within the human brain.
- ✓ Supervised learning methods learn from manually classified training data.
- ✓ Unsupervised learning methods such as Kohonen maps learn without any manual intervention.
- ✓ A system that uses reinforcement learning is given a positive reinforcement when it performs correctly. Credit and blame assignment are important features of such methods.

Neurons

Biological Neurons

The human brain contains over ten billion **neurons**, each of which is connected, on average, to several thousand other neurons. These connections are known as **synapses**, and the human brain contains about 60 trillion such connections.

Neurons are in fact very simple processing elements. Each neuron contains a **soma**, which is the body of the neuron, an **axon**, and a number of **dendrites**. A simplified diagram of a biological neuron is shown in Figure 11.1.

The neuron receives inputs from other neurons along its dendrites, and when this input signal exceeds a certain threshold, the neuron "fires"—in fact, a chemical reaction occurs, which causes an electrical pulse, known as an **action potential**, to be sent down the axon (the output of the neuron), toward synapses that connect the neuron to the dendrites of other neurons.

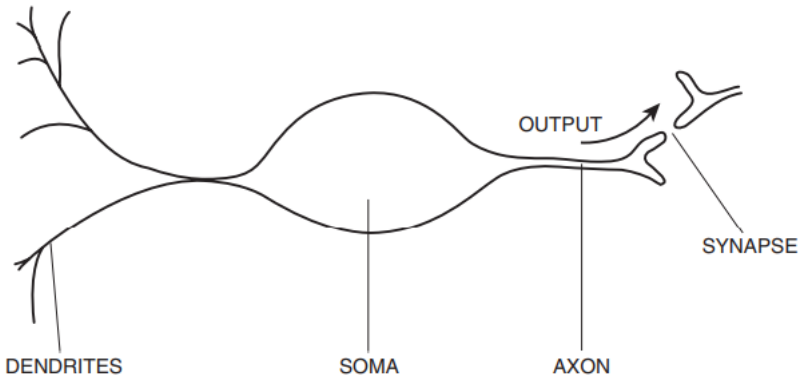

Figure 11.1
A neuron in the human brain

Although each neuron individually is extremely simple, this enormously complex network of neurons is able to process information at a great rate and of extraordinary complexity. The human brain far exceeds in terms of complexity any device created by man, or indeed, any naturally occurring object or structure in the universe, as far as we are aware today.

The human brain has a property known as **plasticity**, which means that neurons can change the nature and number of their connections to other neurons in response to events that occur. In this way, the brain is able to learn. As is explained in Chapter 10, the brain uses a form of credit assignment to strengthen the connections between neurons that lead to correct solutions to problems and weakens connections that lead to incorrect solutions. The strength of a connection, or synapse, determines how much influence it will have on the neurons to which it is connected, and so if a connection is weakened, it will play less of a role in subsequent computations.

Artificial Neurons

Artificial neural networks are modeled on the human brain and consist of a number of artificial neurons. Neurons in artificial neural networks tend to have fewer connections than biological neurons, and neural networks are all (currently) significantly smaller in terms of number of neurons than the human brain.

The neurons that we examine in this chapter were invented by McCulloch and Pitts (1943) and so are often referred to as McCulloch and Pitts neurons.

Each neuron (or **node**) in a neural network receives a number of inputs. A function called the **activation function** is applied to these input values, which results in the **activation level** of the neuron, which is the output value of the neuron. There are a number of possible functions that can be used in neurons. Some of the most commonly used activation functions are illustrated in Figure 11.2.

In Figure 11.2, the *x*-axis of each graph represents the input value to the neuron, and the *y*-axis represents the output, or the activation level, of the neuron.

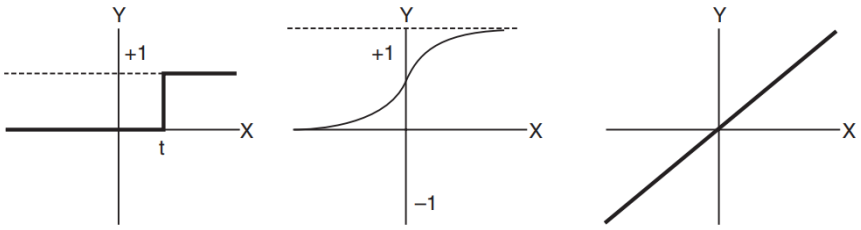

Figure 11.2
Three activation functions

(a) Step function **(b)** Sigmoid function **(c)** Linear function

One of the most commonly used functions is the step function, or **linear threshold function**. In using this function, the inputs to the neuron are summed (having each been multiplied by a weight), and this sum is com- pared with a threshold, *t*. If the sum is greater than the threshold, then the neuron fires and has an activation level of +1. Otherwise, it is inactive and has an activation level of zero. (In some networks, when the sum does not exceed the threshold, the activation level is considered to be —1 instead of 0).

Hence, the behavior of the neuron can be expressed as follows:

$$X = \sum_{i=1}^{n} w_i x_i$$

X is the weighted sum of the *n* inputs to the neuron, x_1 to x_n, where each input, x_n is multiplied by its corresponding weight w_n. For example, let us consider a simple neuron that has just two inputs. Each of these inputs has a weight associated with it, as follows:

$w_1 = 0.8$

$w_2 = 0.4$

The inputs to the neuron are *x*1 and *x*2:

$x_1 = 0.7$

$x_2 = 0.9$

So, the summed weight of these inputs is

$(0.8 \times 0.7) + (0.4 \times 0.9) = 0.92$

The activation level *Y*, is defined for this neuron as

$$Y = \begin{cases} +1 & for\ X > t \\ 0 & for\ X \leq t \end{cases}$$

Hence, if *t* is less than or equal to 0.92, then this neuron will fire with this particular set of inputs. Otherwise, it will have an activation level of zero.

A neuron that uses the linear activation function simply uses the weighted sum of its inputs as its activation level. The sigmoid function converts inputs from a range of -∞ to +∞ into an activation level in the range of 0 to +1.

A neural network consists of a set of neurons that are connected together. Later in this chapter we explore the ways in which neurons are usually connected together. The connections between neurons have weights associated with them, and each neuron passes its output on to the inputs of the neurons to which it is connected. This output depends on the application of the activation function to the inputs it receives. In this way, an input signal to the network is processed by the entire network and an output (or multiple outputs) produced. There is no central processing or control mechanism—the entire network is involved in every piece of computation that takes place.

The way in which neurons behave over time is particularly interesting. When an input is given to a neural network, the output does not appear immediately because it takes some finite period of time for signals to pass from one neuron to another. In artificial neural networks this time is usually very short, but in the human brain, neural connections are surprisingly slow. It is only the enormously parallel nature of the brain that enables it to calculate so quickly.

For neural networks to learn, the weight associated with each connection (equivalent to a synapse in the biological brain) can be changed in response to particular sets of inputs and events. As is mentioned in Chapter 10, Hebbian learning involves increasing the weight of a connection between two neurons if both neurons fire at the same time. We learn more about this later in the chapter.

Perceptrons

The perceptron, which was first proposed by Rosenblatt (1958), is a simple neuron that is used to classify its inputs into one of two categories.

The perceptron can have any number of inputs, which are sometimes arranged into a grid. This grid can be used to represent an image, or a field of vision, and so perceptrons can be used to carry out simple image classification or recognition tasks.

A perceptron uses a step function that returns +1 if the weighted sum of the inputs, *X*, is greater than a threshold, *t*, and —1 if *X* is less than or equal to *t*:

$$X = \sum_{i=1}^{n} w_i x_i$$

$$Y = \begin{cases} +1 & for\ X > t \\ 0 & for\ X \leq t \end{cases}$$

This function is often written as Step (X):

$$Step(X) = \begin{cases} +1 & for\ X > t \\ 0 & for\ X \leq t \end{cases}$$

in which case, the activation function for a perceptron can be written as

$$Y = Step\left(\sum_{i=0}^{n} w_i x_i \right)$$

This function is often written as Step (X):

$$Step(X) = \begin{cases} +1 & for\ X > t \\ 0 & for\ X \leq t \end{cases}$$

in which case, the activation function for a perceptron can be written as

$$Y = Step\left(\sum_{i=0}^{n} w_i x_i \right)$$

Note that here we have allowed i to run from 0 instead of from 1. This means that we have introduced two new variables: w_0 and x_0. We define x_0 as 1, and w_0 as $-t$.

A single perceptron can be used to learn a classification task, where it receives an input and classifies it into one of two categories: 1 or 0. We can consider these to represent *true* and *false*, in which case the perceptron can learn to represent a Boolean operator, such as AND or OR.

The learning process for a perceptron is as follows:

First, random weights are assigned to the inputs. Typically, these weights will be chosen between -0.5 and +0.5.

Next, an item of training data is presented to the perceptron, and its output classification observed. If the output is incorrect, the weights are adjusted to try to more closely classify this input. In other words, if the perceptron incorrectly classifies a positive piece of training data as negative, then the weights need to be modified to increase the output for that set of inputs. This can be done by adding a positive value to the weight of an input that had a negative input value, and vice versa.

The formula for this modification, as proposed by Rosenblatt (Rosenblatt 1960) is as follows:

$$w_i \leftarrow w_i + (a \times x_i \times e)$$

where e is the error that was produced, and a is the **learning rate**, where$0 < a < 1$; e is defined as 0 if the output is correct, and otherwise it is positive if the output is too low and negative if the output is too high. In this way, if the output is too high, a decrease in weight is caused for an input that received a positive value. This rule is known as the **perceptron training rule**.

Once this modification to the weights has taken place, the next piece of training data is used in the same way. Once all the training data have been applied, the process starts again, until all the weights are correct and all errors are zero. Each iteration of this process is known as an **epoch**.

Let us examine a simple example: we will see how a perceptron can learn to represent the logical-OR function for two inputs. We will use a threshold of zero ($t = 0$) and a learning rate of 0.2.

First, the weight associated with each of the two inputs is initialized to a random value between —1 and +1:

$w_1 = -0.2$

$w_2 = 0.4$

Now, the first epoch is run through. The training data will consist of the four combinations of 1's and 0's possible with two inputs.

Hence, our first piece of training data is

x1 = 0

x2 = 0

and our expected output is x1 ∨ x2 = 0.

We apply our formula for Y:

$$Y = Step\left(\sum_{i=0}^{n} w_i x_i\right)$$
$$= Step\big((0 \times -0.2) + (0 \times 0.4)\big)$$
$$= 0$$

Hence, the output Y is as expected, and the error, *e*, is therefore 0. So the weights do not change.

Now, for *x1* = 0 and *x2* = 1:

Y = Step ((0 × —0.2) + (1 × 0.4))
= Step (0.4)
= 1

Again, this is correct, and so the weights do not need to change.

For *x1* = 1 and *x2* = 0:

Y = Step ((1 × - 0.2) + (0 × 0.4))
= Step (- 0.2)
= 0

This is incorrect because 1 ∨ 0 = 1, so we should expect Y to be 1 for this set of inputs. Hence, the weights are adjusted.

We will use the perceptron training rule to assign new values to the weights:

$w_i \leftarrow w_i + (a \times x_i \times e)$

Our learning rate is 0.2, and in this case, the *e* is 1, so we will assign the following value to *w1*:

w1 = - 0.2 + (0.2 × 1 × 1)
= - 0.2 + (0.2)
= 0

We now use the same formula to assign a new value to w2:

w2 = 0.4 + (0.2 × 0 × 1)
= 0.4

Because w2 did not contribute to this error, it is not adjusted.

The final piece of training data is now used (*x1* = 1 and *x2*= 1):

Y = Step ((0 × 1) + (0.4 × 1))
= Step (0 + 0.4)
= Step (0.4)
= 1

This is correct, and so the weights are not adjusted.

This is the end of the first epoch, and at this point the method runs again and continues to repeat until all four pieces of training data are classified correctly.

Table 11.1 A sample run showing how the weights change for a simple perceptron when it learns to represent the logical OR function

Epoch	*X*1	*X*2	Expected *Y*	Actual *Y*	Error	*w*1	*w*2
1	0	0	0	0	0	−0.2	0.4
1	0	1	1	1	0	−0.2	0.4
1	**1**	**0**	**1**	**0**	**1**	**0**	**0.4**
1	1	1	1	1	0	0	0.4
2	0	0	0	0	0	0	0.4
2	0	1	1	1	0	0	0.4
2	**1**	**0**	**1**	**0**	**1**	**0.2**	**0.4**
2	1	1	1	1	0	0.2	0.4
3	0	0	0	0	0	0.2	0.4
3	0	1	1	1	0	0.2	0.4
3	1	0	1	1	0	0.2	0.4
3	1	1	1	1	0	0.2	0.4

Table above shows us complete sequence—it takes just three epochs for the perceptron to correctly learn to classify input values. Lines in which an error was made are marked in **bold**.

After just three epochs, the perceptron learns to correctly model the logical-OR function.

In the same way, a perceptron can be trained to model other logical functions such as AND, but there are some functions that cannot be modeled using a perceptron, such as exclusive OR.

The reason for this is that perceptrons can only learn to model functions that are linearly separable. A **linearly separable** function is one that can be drawn in a two-dimensional graph, and a single straight line can be drawn between the values so that inputs that are classified into one classification are on one side of the line, and inputs that are classified into the other are on the other side of the line. Figure 11.3 shows how such a line can be drawn for the OR function, but not for the exclusive-OR function. Four

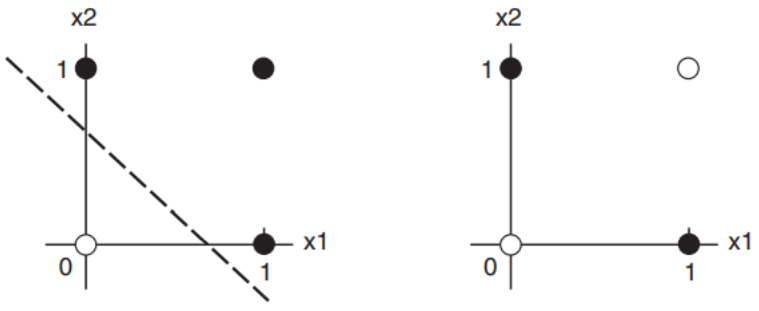

Figure 11.3
Illustrating the difference between a linearly separable function and one which is not

points are plotted on each graph, and a solid dot represents *true*, and a hollow dot represents a value of *false*. It should be clear that no dashed line could be drawn in the second case, for the exclusive OR function, that would separate solid dots from hollow ones.

The reason that a single perceptron can only model functions that are linearly separable can be seen by examining the following function:

$$X = \sum_{i=1}^{n} w_i x_i$$

$$Y = \begin{cases} +1 & for\ X > t \\ -1 & for\ X \leq t \end{cases}$$

Using these functions, we are effectively dividing the search space using a line for which $X = t$. Hence, in a perceptron with two inputs, the line that divides one class from the other is defined as follows:

$$w_1 x_1 + w_2 x_2 = t$$

The perceptron functions by identifying a set of values for w_i, which generates a suitable function. In cases where no such linear function exists, the perceptron cannot succeed.

Multilayer Neural Networks

Most real-world problems are not linearly separable, and so although perceptrons are an interesting model for studying the way in which artificial neurons can work, something more powerful is needed.

As has already been indicated, neural networks consist of a number of neurons that are connected together, usually arranged in layers.

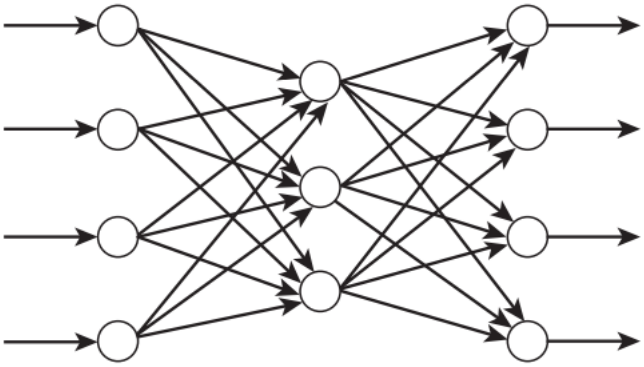

Figure 11.4
A simple three-layer feed-forward neural network

A single perceptron can be thought of as a single-layer perceptron. Multilayer perceptrons are capable of modeling more complex functions, including ones that are not linearly separable, such as the exclusive-OR function.

To see that a multilayer network is capable of modeling a function that is not linearly separable, such as exclusive-OR, note that the functions NOR and NAND are both linearly separable and so can be represented by a single perceptron. By combining these functions together, all other Boolean functions can be generated. Hence, by combining single perceptrons in just two layers, any binary function of two inputs can be generated.

A typical architecture for a multilayer neural network is shown in Figure 11.4.

The network shown in Figure 11.4 is a **feed-forward network**, consisting of three layers.

The first layer is the **input layer**. Each node (or neuron) in this layer receives a single input signal. In fact, it is usually the case that the nodes in this layer are not neurons, but simply act to pass input signals on to the nodes in the next layer, which is in this case a **hidden layer**.

A network can have one or more hidden layers, which contain the neurons that do the real work. Note that each input signal is passed to each of the nodes in this layer and that the output of each node in this layer is passed to each node in the final layer, which is the **output layer**. The output layer carries out the final stage of processing and sends out output signals.

The network is called feed-forward because data are fed forward from the input nodes through to the output nodes. This is in contrast with **recurrent** networks, which we examine in Section 11.5, where some data are passed back from the output nodes to the input nodes.

A typical feed-forward neural network consists of an input layer, one or two hidden layers, and an output layer, and may have anywhere between 10 and 1000 neurons in each layer.

Recurrent Networks

The neural networks we have been studying so far are feed-forward net- works. A feed-forward network is acyclic, in the sense that there are no cycles in the network, because data passes from the inputs to the outputs, and not vice versa,. Once a feed-forward network has been trained, its state is fixed and does not alter as new input data is presented to it. In other words, it does not have **memory**.

A recurrent network can have connections that go backward from output nodes to input nodes and, in fact, can have arbitrary connections between any nodes. In this way, a recurrent network's internal state can alter as sets of input data are presented to it, and it can be said to have a memory.

This is particularly useful in solving problems where the solution depends not just on the current inputs, but on all previous inputs. For example, recurrent networks could be used to predict the stock market price of a particular stock, based on all previous values, or it could be used to predict what the weather will be like tomorrow, based on what the weather has been.

Clearly, due to the lack of memory, feed-forward networks are not able to solve such tasks.

When learning, the recurrent network feeds its inputs through the net- work, including feeding data back from outputs to inputs, and repeats this process until the values of the outputs do not change. At this point, the net- work is said to be in a state of **equilibrium** or **stability**. For this reason, recurrent networks are also known as **attractor networks** because they are attracted to certain output values. The stable values of the network, which are also known as **fundamental memories**, are the output values used as the response to the inputs the network received.

Hence, a recurrent network can be considered to be a **memory**, which is able to learn a set of states—those that act as attractors for it. Once such a network has been trained, for any given input it will output the attractor that is closest to that input.

For example, a recurrent network can be used as an error-correcting net- work. If only a few possible inputs are considered "valid," the network can correct all other inputs to the closest valid input.

It is not always the case that a recurrent network will reach a stable state: some networks are **unstable**, which means they oscillate between different output values.

Unsupervised Learning Networks

The networks we have studied so far in this chapter use **supervised learning**: they are presented with preclassified training data before being asked to classify unseen data. We will now look at a number of methods that are used to enable neural networks to learn in an unsupervised manner.

Kohonen Maps

A **Kohonen map**, or **self-organizing feature map**, is a form of neural net- work invented by Kohonen in the 1980s. The Kohonen map uses the **winner-take-all algorithm**, which leads to a form of unsupervised learning known as **competitive learning**. The winner-take-all algorithm uses the principle that only one neuron provides the output of the network in response to a given input: the neuron that has the highest activation level. During learning, only connections to this neuron have their weights altered.

The purpose of a Kohonen map is to **cluster** input data into a number of clusters. For example, a Kohonen map could be used to cluster news stories into subject categories. A Kohonen map is not told what the categories are: it determines the most useful segmentation itself. Hence, a Kohonen map is particularly useful for clustering data where the clusters are not known in advance.

A Kohonen map has two layers: an input layer and a **cluster layer**, which serves as the output layer. Each input node is connected to every node in the cluster layer, and typically the nodes in the cluster layer are arranged in a grid formation, although this is not essential.

The method used to train a Kohonen map is as follows: Initially, all weights are set to small random values. The learning rate, α, is also set, usually to a small positive value.

An input vector is presented to the input layer of the map. This layer feeds the input data to the cluster layer. The neuron in the cluster layer that most closely matches the input data is declared the winner. This neuron provides the output classification of the map and also has its weights updated.

To determine which neuron wins, its weights are treated as a vector, and this vector is compared with the input vector. The neuron whose weight vector is closest to the input vector is the winner.

The Euclidean distance d_i from the input vector x of a neuron with weight vector w_i is calculated as follows:

$$d_i = \sqrt{\sum_{j=1}^{n} \left(w_{ij} - x_j\right)^2}$$

where n is the number of neurons in the input layer and hence the number of elements in the input vector.

For example, let us calculate the distance between the following two vectors:

$$w_i = \begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix} \quad x = \begin{bmatrix} 3 \\ -1 \\ 2 \end{bmatrix}$$

$$\begin{aligned} \therefore d_i &= \sqrt{(1-3)^2 + (2+1)^2 + (-1-2)^2} \\ &= \sqrt{4+3+9} \\ &= \sqrt{16} \\ &= 4 \end{aligned}$$

So the Euclidean distance between these two vectors is 4.

The neuron for which d_i is the smallest is the winner, and this neuron has its weight vector updated as follows:

$$w_{ij} \leftarrow w_{ij} + \alpha\left(x_j - w_{ij}\right)$$

This adjustment moves the weight vector of the winning neuron closer to the input vector that caused it to win.

In fact, rather than just the winning neuron having its weights updated, a neighborhood of neurons around the winner are usually updated. The neighborhood is usually defined as a radius within the two-dimensional grid of neurons around the winning neuron.

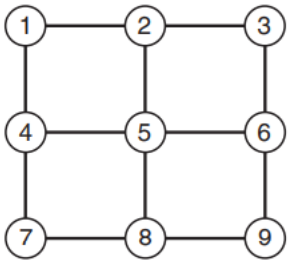

Figure 11.5
The cluster layer of a simple Kohonen map

Typically, the radius decreases over time as the training data are examined, ending up fixed at a small value. Similarly, the learning rate is often reduced during the training phase.

This training phase usually terminates when the modification of weights becomes very small for all the cluster neurons. At this point, the network has extracted from the training data a set of clusters, where similar items are contained within the same cluster, and similar clusters are near to each other.

Kohonen Map Example

Let us examine a simplified example of a Kohonen map.

Our Kohonen map has just two inputs and nine cluster neurons, which are arranged into a 3×3 grid, as shown in Figure 11.5.

Figure 11.5 shows how the neurons are arranged in a grid. Each node in the cluster layer is connected to each of the two input nodes. The cluster layer nodes are not connected to each other. The grid shown in Figure 11.5 does not represent physical connection, but rather spatial proximity—node 1 is close to nodes 2 and 4. This spatial proximity of neurons is used to calculate the neighborhood set that is used to determine which weights to update during the training phase.

Note that this square arrangement is by no means necessary. The nodes are often arranged in a rectangular grid, but other shapes can be used equally successfully.

Because there are two input nodes in the network, we can represent each input as a position in two-dimensional space. Figure 11.6 shows the nine input values that are to be used to train this network.

In Figure 11.6, x1 and x2 are the two input values that are to be presented to the input layer, which contains two neurons.

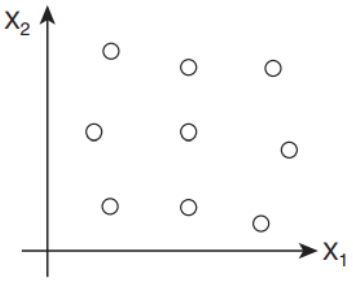

Figure 11.6
Training data for the Kohonen map shown in Figure 11.5

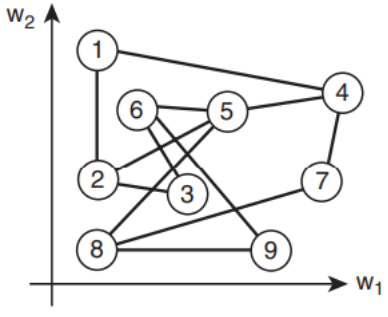

Figure 11.7
Initial weight vectors for the Kohonen map

Note that the training data have been selected randomly from the available space, such that they fill as much of the space as possible. In this way the data will be as representative as possible of all available input data, and so the Kohonen map will be able to cluster the input space optimally.

Because each neuron in the cluster layer has connections to the two input layer neurons, their weight vectors can be plotted in two-dimensional space. These weight vectors are initially set to random values, which are shown in Figure 11.7. The connections between nodes in Figure 11.7 represent spatial proximity again, as in Figure 11.5.

Because there are nine cluster nodes and nine pieces of training data, we expect the network to assign each neuron to one piece of training data. Most real Kohonen maps consist of far more neurons, and many more training data are usually used.

In our simple example, by running a number of iterations of the Kohonen map, the weight vectors are modified to those shown in Figure 11.8.

In this case, it is easy to see what the map has done: by modifying the weight vector of each neuron so that it closely resembles one training vector, the nodes have been modified so that each node will respond extremely well to one of the input data. When a new piece of input data is presented, it will be classified by the node whose weight vector is closest to it. Additionally, that node's weight vector will be moved slightly toward the new piece of input data. In this way, the network continues to learn as new data are presented to it. By decreasing the learning rate over time, the network can be forced to reach a stable state where the weights no longer change, or change only very slightly, when presented with new input data.

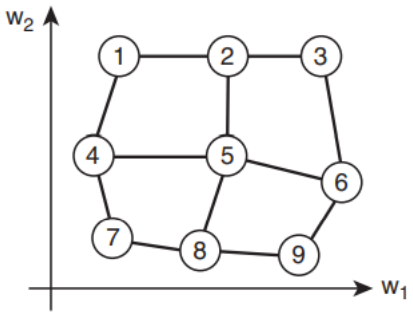

Figure 11.8
Weight vectors after training the Kohonen map

This example illustrates the self-organizing nature of Kohonen maps. The space-filling shape shown in Figure 11.8 is typical of the behavior of these networks.

Evolving Neural Networks

The ideas that we cover in Chapter 14 on genetic algorithms can be applied to neural networks. Genetic algorithms can be used to evolve suitable starting weight vectors for a network. This is useful because the initial weight vector that is chosen for a network can significantly affect the ability of the network to solve a particular problem. Neural networks suffer from many of the problems faced by search methods presented in Part 2 of this book, such as falling into local minima. By repeatedly running a full training session on a neural network with different random starting weights, this problem can be avoided. Clearly, this problem can also be avoided by using evolutionary methods to select starting weight vectors.

Similarly, a genetic algorithm can be used to determine the connectivity of the network. In this way, the number of neurons and the connections between those neurons can be evolved to produce an optimal architecture.