CHAPTER 14

Unsupervised Learning

Many of the learning algorithms that we have seen to date have made use of a training set that consists of a collection of labelled target data, or at least (for evolutionary and reinforcement learning) some scoring system that identifies whether or not a prediction is good or not. Targets are obviously useful, since they enable us to show the algorithm the correct answer to possible inputs, but in many circumstances they are difficult to obtain—they could, for instance, involve somebody labelling each instance by hand. In addition, it doesn't seem to be very biologically plausible: most of the time when we are learning, we don't get told exactly what the right answer should be. In this chapter we will consider exactly the opposite case, where there is no information about the correct outputs available at all, and the algorithm is left to spot some similarity between different inputs for itself.

Unsupervised learning is a conceptually different problem to supervised learning. Obviously, we can't hope to perform regression: we don't know the outputs for any points, so we can't guess what the function is. Can we hope to do classification then? The aim of classification is to identify similarities between inputs that belong to the same class. There isn't any information about the correct classes, but if the algorithm can exploit similarities between inputs in order to cluster inputs that are similar together, this might perform classification automatically. So the aim of unsupervised learning is to find clusters of similar inputs in the data without being explicitly told that these datapoints belong to one class and those to a different class. Instead, the algorithm has to discover the similarities for itself. We have already seen some unsupervised learning algorithms in Chapter 6, where the focus was on dimensionality reduction, and hence clustering of similar datapoints together.

The supervised learning algorithms that we have discussed so far have aimed to minimise some external error criterion—mostly the sum-of-squares error—based on the difference between the targets and the outputs. Calculating and minimising this error was possible because we had target data to calculate it from, which is not true for unsupervised learning. This means that we need to find something else to drive the learning. The problem is more general than sum-of-squares error: we can't use any error criterion that relies on targets or other outside information (an external error criterion), we need to find something internal to the algorithm. This means that the measure has to be independent of the task, because we can't keep on changing the whole algorithm every time a new task is introduced. In supervised learning the error criterion was task-specific, because it was based on the target data that we provided.

To see how to work out a general error criterion that we can use, we need to go back to some of the important concepts that were discussed in Section 2.1.1: input space and weight space. If two inputs are close together then it means that their vectors are similar, and so the distance between them is small (distance measures were discussed in Section 7.2.3, but

here we will stick to Euclidean distance). Then inputs that are close together are identified as being similar, so that they can be clustered, while inputs that are far apart are not clustered together. We can extend this to the nodes of a network by aligning weight space with input space. Now if the weight values of a node are similar to the elements of an input vector then that node should be a good match for the input, and any other inputs that are similar. In order to start to see these ideas in practice we'll look at a simple clustering algorithm, the k-Means Algorithm, which has been around in statistics for a long time.

14.1 THE K-MEANS ALGORITHM

If you have ever watched a group of tourists with a couple of tour guides who hold umbrellas up so that everybody can see them and follow them, then you have seen a dynamic version of the k-means algorithm. Our version is simpler, because the data (playing the part of the tourists) does not move, only the tour guides.

Suppose that we want to divide our input data into k categories, where we know the value of k (for example, we have a set of medical test results from lots of people for three diseases, and we want to see how well the tests identify the three diseases). We allocate k cluster centres to our input space, and we would like to position these centres so that there is one cluster centre in the middle of each cluster. However, we don't know where the clusters are, let alone where their 'middle' is, so we need an algorithm that will find them. Learning algorithms generally try to minimise some sort of error, so we need to think of an error criterion that describes this aim. The idea of the 'middle' is the first thing that we need to think about. How do we define the middle of a set of points? There are actually two things that we need to define:

A distance measure In order to talk about distances between points, we need some way to measure distances. It is often the normal Euclidean distance, but there are other alternatives; we've covered some other alternatives in Section 7.2.3.

The mean average Once we have a distance measure, we can compute the central point of a set of datapoints, which is the mean average (if you aren't convinced, think what the mean of two numbers is, it is the point halfway along the line between them). Actually, this is only true in Euclidean space, which is the one you are used to, where everything is nice and flat. Everything becomes a lot trickier if we have to think about curved spaces; when we have to worry about curvature, the Euclidean distance metric isn't the right one, and there are at least two different definitions of the mean. So we aren't going to worry about any of these things, and we'll assume that space is flat. This is what statisticians do all the time.

We can now think about a suitable way of positioning the cluster centres: we compute the mean point of each cluster, $\boldsymbol{\mu}_{c(i)}$, and put the cluster centre there. This is equivalent to minimising the Euclidean distance (which is the sum-of-squares error again) from each datapoint to its cluster centre.

How do we decide which points belong to which clusters? It is important to decide, since we will use that to position the cluster centres. The obvious thing is to associate each point with the cluster centre that it is closest too. This might change as the algorithm iterates, but that's fine.

We start by positioning the cluster centres randomly through the input space, since we don't know where to put them, and then we update their positions according to the data. We decide which cluster each datapoint belongs to by computing the distance between each

datapoint and all of the cluster centres, and assigning it to the cluster that is the closest. Note that we can reduce the computational cost of this procedure by using the KD-Tree algorithm that was described in Section 7.2.2. For all of the points that are assigned to a cluster, we then compute the mean of them, and move the cluster centre to that place. We iterate the algorithm until the cluster centres stop moving. Here is the algorithmic description:

The k-Means Algorithm

- **Initialisation**
  - choose a value for k
  - choose k random positions in the input space
  - assign the cluster centres $\boldsymbol{\mu}_j$ to those positions
- **Learning**
  - repeat
    - for each datapoint $\mathbf{x}_i$:
      - compute the distance to each cluster centre
      - assign the datapoint to the nearest cluster centre with distance

$$d_i = \min_j d(\mathbf{x}_i, \boldsymbol{\mu}_j). \tag{14.1}$$

    - for each cluster centre:
      - move the position of the centre to the mean of the points in that cluster (N_j is the number of points in cluster j):

$$\boldsymbol{\mu}_j = \frac{1}{N_j} \sum_{i=1}^{N_j} \mathbf{x}_i \tag{14.2}$$

  - until the cluster centres stop moving
- **Usage**
  - for each test point:
    - compute the distance to each cluster centre
    - assign the datapoint to the nearest cluster centre with distance

$$d_i = \min_j d(\mathbf{x}_i, \boldsymbol{\mu}_j). \tag{14.3}$$

The NumPy implementation follows these steps almost exactly, and we can take advantage of the `np.argmin()` function, which returns the index of the minimum value, to find the closest cluster. The code that computes the distances, finds the nearest cluster centre, and updates them can then be written as:

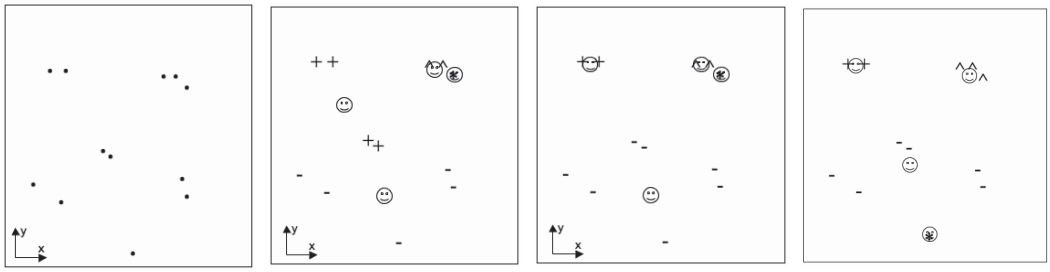

FIGURE 14.1 *Left:* A two-dimensional dataset. *Right:* Three possible ways to position 4 centres (drawn as faces) using the k-means algorithm, which is clearly susceptible to local minima.

```
# Compute distances
distances = np.ones((1,self.nData))*np.sum((data-self.centres[0,:])**2,
axis=1)
for j in range(self.k-1):
    distances = np.append(distances,np.ones((1,self.nData))*np.sum((data-
    self.centres[j+1,:])**2,axis=1),axis=0)

# Identify the closest cluster
cluster = distances.argmin(axis=0)
cluster = np.transpose(cluster*np.ones((1,self.nData)))

# Update the cluster centres
for j in range(self.k):
    thisCluster = np.where(cluster==j,1,0)
    if sum(thisCluster)>0:
        self.centres[j,:] = np.sum(data*thisCluster,axis=0)/np.sum(
        thisCluster)
```

To see how this works in practice, Figures 14.1 and 14.2 show some data and some different ways to cluster that data computed by the k-means algorithm. It should be clear that the algorithm is susceptible to local minima: depending upon where the centres are initially positioned in the space, you can get very different solutions, and many of them look very unlikely to our eyes. Figure 14.2 shows examples of what happens when you choose the number of centres wrongly. There are certainly cases where we don't know in advance how many clusters we will see in the data, but the k-means algorithm doesn't deal with this at all well.

At the cost of significant extra computational expense, we can get around both of these problems by running the algorithm many different times. To find a good local optimum (or even the global one) we use many different initial centre locations, and the solution that minimises the overall sum-of-squares error is likely to be the best one.

By running the algorithm with lots of different values of k, we can see which values give us the best solution. Of course, we need to be careful with this. If we still just measure the sum-of-squares error between each datapoint and its nearest cluster centre, then when

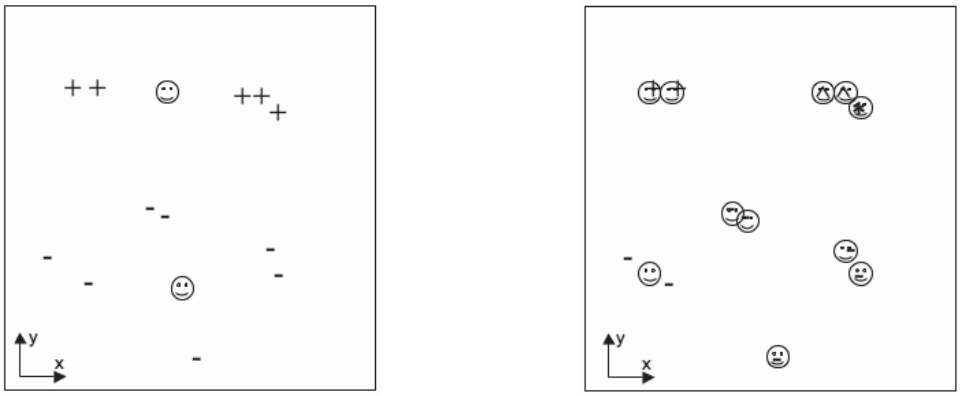

FIGURE 14.2 *Left:* A solution with only 2 classes, which does not match the data well. *Right:* A solution with 11 classes, showing severe overfitting.

we set k to be equal to the number of datapoints, we can position one centre on every datapoint, and the sum-of-squares error will be zero (in fact, this won't happen, since the random initialisation will mean that several clusters will end up coinciding). However, there is no generalisation in this solution: it is a case of serious overfitting. However, by computing the error on a validation set and multiplying the error by k we can see something about the benefit of adding each extra cluster centre.

14.1.1 Dealing with Noise

There are lots of reasons for performing clustering, but one of the more common ones is to deal with noisy data readings. These might be slightly corrupted, or occasionally just plain wrong. If we can choose the clusters correctly, then we have effectively removed the noise, because we replace each noisy datapoint by the cluster centre (we will use this way of representing datapoints for other purposes in Section 14.2). Unfortunately, the mean average, which is central to the k-means algorithm, is very susceptible to outliers, i.e., very noisy measurements. One way to avoid the problem is to replace the mean average with the median, which is what is known as a robust statistic, meaning that it is not affected by outliers (the mean of $(1, 2, 1, 2, 100)$ is 21.2, while the median is 2). The only change that is needed to the algorithm is to replace the computation of the mean with the computation of the median. This is computationally more expensive, as we've discussed previously, but it does remove noise effectively.

14.1.2 The k-Means Neural Network

The k-means algorithm clearly works, despite its problems with noise and the difficulty with choosing the number of clusters. Interestingly, while it might seem a long way from neural networks, it isn't. If we think about the cluster centres that we optimise the positions of as locations in weight space, then we could position neurons in those places and use neural network training. The computation that happened in the k-means algorithm was that each input decided which cluster centre it was closest to by calculating the distance to all of the centres. We could do this inside a neural network, too: the location of each neuron is its position in weight space, which matches the values of its weights. So for each input, we

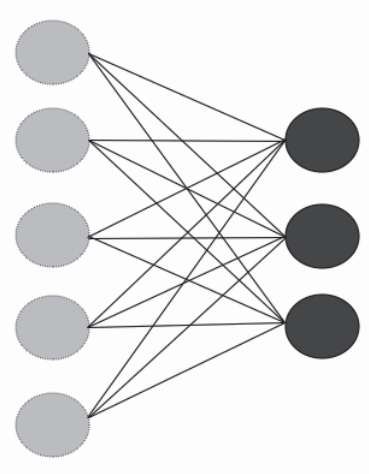

FIGURE 14.3 A single-layer neural network can implement the k-means solution.

just make the activation of a node be the distance between that node in weight space and the current input, as we did for Radial Basis Functions in Chapter 5. Then training is just moving the position of the node, which means adjusting the weights.

So, we can implement the k-means algorithm using a set of neurons. We will use just one layer of neurons, together with some input nodes, and no bias node. The first layer will be the inputs, which don't do any computation, as usual, and the second layer will be a layer of competitive neurons, that is, neurons that 'compete' to fire, with only one of them actually succeeding. Only one cluster centre can represent a particular input vector, and so we will choose the neuron with the highest activation h to be the one that fires. This is known as winner-takes-all activation, and it is an example of competitive learning, since the set of neurons compete with each other to fire, with the winner being the one that best matches (i.e., is closest to) the input. Competitive learning is sometimes said to lead to grandmother cells, because each neuron in the network will learn to recognise one particular feature, and will fire only when that input is seen. You would then have a specific neuron that was trained to recognise your grandmother (and others for anybody else/anything else that you see often).

We will choose k neurons (for hopefully obvious reasons) and fully connect the inputs to the neurons, as usual. There is a picture of this network in Figure 14.3. We will use neurons with a linear transfer function, computing the activation of the neurons as simply the product of the weights and inputs:

$$h_i = \sum_j w_{ij} x_j. \tag{14.4}$$

Providing that the inputs are normalised so that their absolute size is the same (a point that we'll come back to in Section 14.1.3), this effectively measures the distance between the input vector and the cluster centre represented by that neuron, with larger numbers (higher activations) meaning that the two points are closer together.

So the winning neuron is the one that is closest to the current input. The question is how can we then change the position of that neuron in weight space, that is, how do we update its weights? In the k-means algorithm that was described earlier it was easy: we just set the cluster centre to be the mean of all the datapoints that were assigned to that

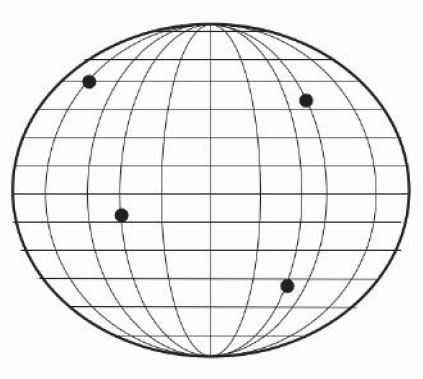

FIGURE 14.4 A set of neurons positioned on the unit sphere in 3D.

centre. However, when we do neural network training, we often feed in just one input vector at a time and change the weights (that is, we use the algorithm on-line, rather than batch). We therefore do not know the mean because we don't know about all the datapoints, just the current one. So we approximate it by moving the winning neuron closer to the current input, making that centre even more likely to be the best match next time that input is seen. This corresponds to:

$$\Delta w_{ij} = \eta x_j. \tag{14.5}$$

However, this is not good enough. To see why not, let's get back to that question of normalisation. This is important enough to need its own subsection.

14.1.3 Normalisation

Suppose that the weights of all the neurons are small (maybe less than 1) except for those to one particular neuron. We'll make those weights be 10 for the example. If an input vector with values $(0.2, 0.2, -0.1)$ is presented, and it happens to be an exact match for one of the neurons, then the activation of that neuron will be $0.2 \times 0.2 + 0.2 \times 0.2 + -0.1 \times -0.1 = 0.09$. The other neurons are not perfect matches, so their activations should all be less. However, consider the neuron with large weights. Its activation will be $10 \times 0.2 + 10 \times 0.2 + 10 \times -0.1 = 3$, and so it will be the winner. Thus, we can only compare activations if we know that the weights for all of the neurons are the same size. We do this by insisting that the weight vector is normalised so that the distance between the vector and the origin (the point $(0, 0, \ldots 0)$) is one. This means that all of the neurons are positioned on the unit hypersphere, which we described in Section 2.1.2 when we talked about the curse of dimensionality: it is the set of all points that are distance one from the origin, so it is a circle in 2D, a sphere in 3D (as shown in Figure 14.4), and a hypersphere in higher dimensions.

Computing this normalisation in NumPy takes a little bit of care because we are normalising the total Euclidean distance from the origin, and the sum and division are row-wise rather than column-wise, which means that the matrix has to be transposed before and after the division:

```
normalisers = np.sqrt(np.sum(data**2,axis=1))*np.ones((1,shape(data)[0]))
data = np.transpose(np.transpose(data)/normalisers)
```

The neuronal activation (Equation (14.4)) can be written as:

$$h_i = \mathbf{W}_i^T \cdot \mathbf{x}, \tag{14.6}$$

where, as usual, $\cdot$ refers to the inner product or scalar product between the two vectors, and $\mathbf{W}_i^T$ is the transpose of the ith row of W. The inner product computes $\|\mathbf{W}_i\|\|\mathbf{x}\| \cos\theta$, where θ is the angle between the two vectors and $\|\cdot\|$ is the magnitude of the vector. So if the magnitude of all the vectors is one, then only the angle θ affects the size of the dot product, and this tells us about the difference between the vector directions, since the more they point in the same direction, the larger the activation will be.

14.1.4 A Better Weight Update Rule

The weight update rule given in Equation (14.5) lets the weights grow without any bound, so that they do not lie on the unit hypersphere any more. If we normalise the inputs as well, which certainly seems reasonable, then we can use the following weight update rule:

$$\Delta w_{ij} = \eta(x_j - w_{ij}), \tag{14.7}$$

which has the effect of moving the weight w_{ij} directly towards the current input. Remember that the only weights that we are updating are those of the winning unit:

```
for i in range(self.nEpochs):
    for j in range(self.nData):
        activation = np.sum(self.weights*np.transpose(data[j:j+1,:]),axis=0)
        winner = np.argmax(activation)
        self.weights[:,winner] += self.eta * data[j,:] - self.weights[:,
        winner]
```

For many of our supervised learning algorithms we minimised the sum-of-squares difference between the output and the target. This was a global error criterion that affected all of the weights together. Now we are minimising a function that is effectively independent in each weight. So the minimisation that we are doing is actually more complicated, even though it doesn't look it. This makes it very difficult to analyse the behaviour of the algorithm, which is a general problem for competitive learning algorithms. However, they do tend to work well.

Now that we have a weight update rule that works, we can consider the entire algorithm for the on-line k-means network:

The On-Line k-Means Algorithm

- **Initialisation**
  - choose a value for k, which corresponds to the number of output nodes
  - initialise the weights to have small random values
- **Learning**
  - normalise the data so that all the points lie on the unit sphere
  - repeat:
    - for each datapoint:
      - compute the activations of all the nodes
      - pick the winner as the node with the highest activation
      - update the weights using Equation (14.7)
    - until number of iterations is above a threshold
- **Usage**
  - for each test point:
    - compute the activations of all the nodes
    - pick the winner as the node with the highest activation

14.1.5 Example: The Iris Dataset Again

Now that we have a method of training the k-means algorithm we can use it to learn about data. Except we need to think about how to understand the results. If there aren't any labels in the data, then we can't really do much to analyse the results, since we don't have anything to compare them with. However, we might use unsupervised learning methods to cluster data where we know at least some of the labels. For example, we can use the algorithm on the iris dataset that we looked at in Section 4.4.3, where we classified three types of iris flowers using the MLP. All we need to do is to give some of the data to the algorithm and train it, and then use some more to test the output. However, the output of the algorithm isn't as clear now, because we don't use the labels that come with the data, since we aren't doing supervised learning anymore. To get around that, we need to work out some way of turning the results from the algorithm, which is the index of the cluster that best matches it, into a classification output that we can compare with the labels. This is relatively easy if we used three clusters in the algorithm, since there should hopefully be a one-to-one correspondence between them, but it might turn out that using more clusters gets better results, although this will make the analysis more difficult. You can do this by hand if there are relatively small numbers of datapoints, or you could use a supervised learning algorithm to do it for you, as is discussed next.

To see how the k-means algorithm is used, we can see how it is used on the iris dataset:

```
import kmeansnet
net = kmeansnet.kmeans(3,train)
net.kmeanstrain(train)
cluster = net.kmeansfwd(test)
print cluster
print iris[3::4,4]
```

The output that is produced by this in an example run is (where the top line is the output of the algorithm and the bottom line is the classes from the dataset):

```
[ 0. 0. 0. 0. 0. 1. 1. 1. 1. 2. 1. 2. 2. 2. 0. 1. 2. 1. 0.
  1. 2. 2. 2. 1. 1. 2. 0. 0. 1. 0. 0. 0. 0. 2. 0. 2. 1.]
[ 1. 1. 1. 1. 1. 2. 2. 2. 1. 0. 2. 0. 0. 0. 1. 1. 0. 2. 2.
  2. 0. 0. 0. 2. 2. 0. 1. 2. 1. 1. 1. 1. 1. 0. 1. 0. 2.]
```

and then we can see that cluster 0 corresponds to label 1 and cluster 1 to label 2, in which case the algorithm gets 1 of cluster 0 wrong, 2 of cluster 1, and none of cluster 2.

14.1.6 Using Competitive Learning for Clustering

Deciding which cluster any datapoint belongs to is now an easy task: we present it to the trained algorithm and look what is activated. If we don't have any target data, then the problem is finished. However, for many problems we might want to interpret the best-matching cluster as a class label (alternatively, a set of cluster centres could all correspond to one class). This is fine, since if we have target data we can match the output classes to the targets, provided that we are a bit careful: there is no reason why the order of the nodes in the network should match the order in the data, since the algorithm knows nothing about that order. For that reason, when assigning class labels to the outputs, you need to check which numbers match up carefully, or the results will look a lot worse than they actually are.

There is an alternative solution to this problem of assigning labels, and it is one that we have seen before. In Chapter 5 we considered using the k-means network in order to train the positions of the RBF nodes. It is now possible to see how this works. The k-means part positions the RBFs in the input space, so that they represent the input data well. A Perceptron is then used on top of this in order to provide the match to the outputs in the supervised learning part of the network. Since this is now supervised learning, it ensures that the output categories match the target data classes. It also means that you can use lots of clusters in the k-means network without having to work out which datapoints belong to which cluster, since the Perceptron will do this for you.

We are now going to look at another major algorithm in competitive learning, the Self-Organising Feature Map. As motivation for it, we are going to consider a sample problem for competitive learning, which is a problem in data compression called vector quantisation.

14.2 VECTOR QUANTISATION

We've already discussed using competitive learning for removing noise. There is a related application, data compression, which is used both for storing data and for the transmission of speech and image data. The reason that the applications are related is that both replace the current input by the cluster centre that it belongs to. For noise reduction we do this to replace the noisy input with a cleaner one, while for data compression we do it to reduce the number of datapoints that we send.

Both of these things can be understood by considering them as examples of data communication. Suppose that I want to send data to you, but that I have to pay for each data bit I transmit, so I want to keep the amount of data that I send to a minimum. I notice that there are lots of repeated datapoints, so I decide to encode my data before I send it, so that instead of sending the entire set, we agree on a codebook of prototype vectors together. Now, instead of transmitting the actual data, I can transmit the index of that datapoint in the codebook, which is shorter. All you have to do is take the indices I send you and look them up, and you have the data. We can actually make the code even more efficient by using shorter indices for the datapoints that are more common. This is an important problem in information theory, and every kind of sound and image compression algorithm has a different method of solving it.

There is one problem with the scenario so far, which is that the codebook won't contain every possible datapoint. What happens when I want to send a datapoint and it isn't in the codebook? In that case we need to accept that our data will not look exactly the same, and I send you the index of the prototype vector that is closest to it (this is known as vector quantisation, and is the way that lossy compression works).

Figure 14.5 shows an interpretation of prototype vectors in two dimensions. The dots at the centre of each cell are the prototype vectors, and any datapoint that lies within a cell is represented by the dot. The name for each cell is the Voronoi set of a particular prototype. Together, they produce the Voronoi tesselation of the space. If you connect together every pair of points that share an edge, as is shown by the dotted lines, then you get the Delaunay triangulation, which is the optimal way to organise the space to perform function approximation.

The question is how to choose the prototype vectors, and this is where competitive learning comes in. We need to choose prototype vectors that are as close as possible to all of the possible inputs that we might see. This application is called learning vector quantisation because we are learning an efficient vector quantisation. The k-means algorithm can be used to solve the problem if we know how large we want our codebook to be. However, another algorithm turns out to be more useful, the Self-Organising Feature Map, which is described next.

14.3 THE SELF-ORGANISING FEATURE MAP

By far the most commonly used competitive learning algorithm is the Self-Organising Feature Map (often abbreviated to SOM), which was proposed by Teuvo Kohonen in 1988. Kohonen was considering the question of how sensory signals get mapped into the cerebral cortex of the brain with an order. For example, in the auditory cortex, which deals with the sounds that we hear, neurons that are excited (i.e., that are caused to fire) by similar sounds are positioned closely together, whereas two neurons that are excited by very different sounds will be far apart.

There are two novel departures in this for us: firstly, the relative locations of the neurons in the network matters (this property is known as feature mapping—nearby neurons

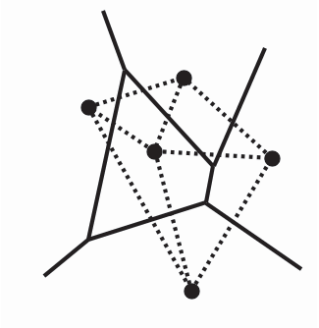

FIGURE 14.5 The Voronoi tesselation of space that performs vector quantisation. Any datapoint is represented by the dot within its cell, which is the prototype vector.

correspond to similar input patterns), and secondly, the neurons are arranged in a grid with connections between the neurons, rather than in layers with connections only between the different layers. In the auditory cortex there appears to be sheets of neurons arranged in 2D, and that is the typical arrangement of neurons for the SOM: a grid of neurons arranged in 2D, as can be seen in Figure 14.6. A 1D line of neurons is also sometimes used. In mathematical terms, the SOM demonstrates relative ordering preservation, which is sometimes known as topology preservation. The relative ordering of the inputs should be preserved by the ordering in the neurons, so that neurons that are close together represent inputs that are close together, while neurons that are far apart represent inputs that are far apart.

This topology preservation is not necessarily possible, because the SOM typically uses a 1D or 2D array of neurons, and most of our input spaces are of much higher dimensionality than that. This means that the ordering cannot be preserved. We have seen this in Figure 1.2, where one view of some wind turbines made it look like they are on top of each other, when they clearly are not, because we used a two-dimensional representation of three-dimensional reality. You've probably seen the same thing in other photos, where trees appear to be growing out of somebody's head. A different way to see the same thing is given in Figure 14.7, where mismatches between the topology of the input space and map lead to changes in the relative ordering. The best that can be said is that SOM is perfectly topology-preserving, which means that if the dimensionality of the input and the map correspond, then the topology of the input space will be preserved. We are going to look at other methods of performing dimensionality reduction in Chapter 6.

The question, then, is how we can implement feature mapping in an unsupervised learning algorithm. The first thing to recognise is that we need some interaction between the neurons in the network, so that when one neuron fires, it affects what happens to those around it. We have seen something like this before, for example, between different layers of the MLP, but now we are thinking about neurons that are within a layer. These are known as lateral connections (i.e., within the layer of the network). How should this interaction work? We are trying to introduce feature mapping, so neurons that are close together in the map should represent similar features. This means that the winning neuron should pull other neurons that are close to it in the network closer to itself in weight space, which means that we need positive connections. Likewise, neurons that are further away should represent different features, and so should be a long way off in weight space, so the winning neuron 'repels' them, by using negative connections to push them away. Neurons that are very far away in the network should already represent different features, so we just ignore them.

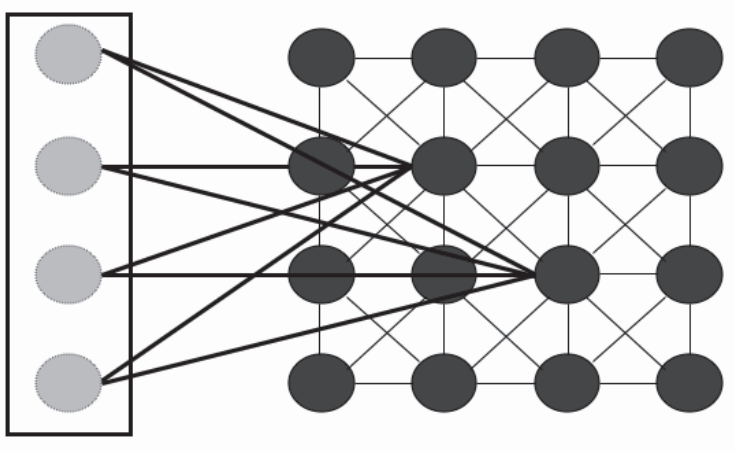

FIGURE 14.6 The Self-Organising Map network. As usual, input nodes (on the left) do no computation, and the weights are modified to change the activations of the neurons (weights are only shown to two nodes for clarity). However, the nodes within the SOM affect each other in that the winning node also changes the weights of neurons that are close to it. Connections are shown in the figure to the eight closest nodes, but this is a parameter of the network.

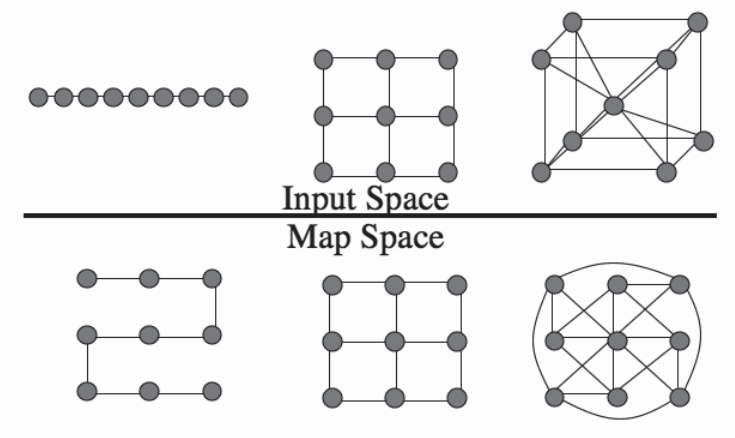

FIGURE 14.7 When inputs in 1D (a straight line), a 2D grid, and a 3D cube are represented by a 2D grid of neurons, the relative ordering is not perfectly preserved. The 1D line is bent, which means that points that used to be a long way apart (such as the first and sixth on the line) are now close together, while the cube becomes very complicated. The lines in the bottom part of the figure represent connections that are meant to be close.

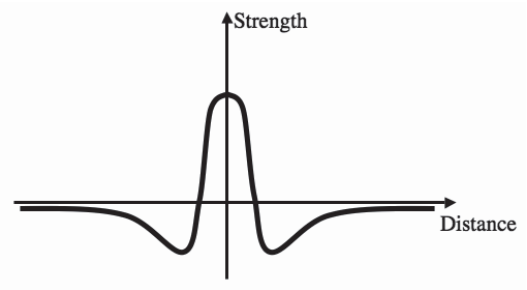

FIGURE 14.8 Graph of the strength of lateral connections for a feature mapping algorithm known as the 'Mexican Hat'.

This is known as the 'Mexican Hat' form of lateral connections, for reasons that should be clear from the picture in Figure 14.8. We can then just use ordinary competitive learning, just like we did for the k-means network in Section 14.1.2. The Self-Organising Map does pretty much exactly this.

14.3.1 The SOM Algorithm

Using the full Mexican hat lateral interactions between neurons is fine, but it isn't essential. In Kohonen's SOM algorithm, the weight update rule is modified instead, so that information about neighbouring neurons is included in the learning rule, which makes the algorithm simpler. The algorithm is a competitive learning algorithm, so that one neuron is chosen as the winner, but when its weights are updated, so are those of its neighbours, although to a lesser extent. Neurons that are not within the neighbourhood are ignored, not repelled.

We will now look at the SOM algorithm before examining some of the details further.

The Self-Organising Feature Map Algorithm

- **Initialisation**
  - choose a size (number of neurons) and number of dimensions d for the map
  - either:
    - choose random values for the weight vectors so that they are all different OR
    - set the weight values to increase in the direction of the first d principal components of the dataset
- **Learning**
  - repeat:
    - for each datapoint:
      - select the best-matching neuron n_b using the minimum Euclidean distance between the weights and the input,

$$n_b = \min_j \|\mathbf{x} - \mathbf{w}_j^T\|. \tag{14.8}$$

    - update the weight vector of the best-matching node using:

$$\mathbf{w}_j^T \leftarrow \mathbf{w}_j^T + \eta(t)(\mathbf{x} - \mathbf{w}_j^T), \tag{14.9}$$

where $\eta(t)$ is the learning rate.

* update the weight vector of all other neurons using:

$$\mathbf{w}_j^T \leftarrow \mathbf{w}_j^T + \eta_n(t)h(n_b, t)(\mathbf{x} - \mathbf{w}_j^T), \tag{14.10}$$

where $\eta_n(t)$ is the learning rate for neighbourhood nodes, and $h(n_b, t)$ is the neighbourhood function, which decides whether each neuron should be included in the neighbourhood of the winning neuron (so $h = 1$ for neighbours and $h = 0$ for non-neighbours)

* reduce the learning rates and adjust the neighbourhood function, typically by $\eta(t+1) = \alpha\eta(t)^{k/k_{\max}}$ where $0 \leq \alpha \leq 1$ decides how fast the size decreases, k is the number of iterations the algorithm has been running for, and $k_{\max}$ is when you want the learning to stop. The same equation is used for both learning rates (η, η_n) and the neighbourhood function $h(n_b, t)$.

– until the map stops changing or some maximum number of iterations is exceeded

- **Usage**

– for each test point:

* select the best-matching neuron n_b using the minimum Euclidean distance between the weights and the input:

$$n_b = \min_j \|\mathbf{x} - \mathbf{w}_j^T\| \tag{14.11}$$

14.3.2 Neighbourhood Connections

The size of the neighbourhood is thus another parameter that we need to control. How large should the neighbourhood of a neuron be? If we start our network off with random weights, as we did for the MLP, then at the beginning of learning, the network is pretty well unordered (as the weights are random, two nodes that are very close in weight space could be on opposite sides of the map, and vice versa) and so it makes sense that the neighbourhoods should be large, so that we get the rough ordering of the network correct. However, once the network has been learning for a while, the rough ordering has already been created, and the algorithm starts to fine-tune the individual local regions of the network. At this stage, the neighbourhoods should be small, as is shown in Figure 14.9. It therefore makes sense to reduce the size of the neighbourhood as the network adapts. These two phases of learning are also known as ordering and convergence. Typically, we reduce the neighbourhood size by a small amount at each iteration of the algorithm. We control the learning rate η in exactly the same way, so that it starts off large and decreases over time, as is shown in the algorithm below.

The fact that the size of the neighbourhood changes as the algorithm runs has consequences for an implementation. There is no point using actual connections between nodes, since the number of these will change as the algorithm runs. We therefore set up a matrix that measures the distances between nodes in the network and choose the nodes in the neighbourhood of a particular node as those within a neighbourhood radius that shrinks as the algorithm runs.

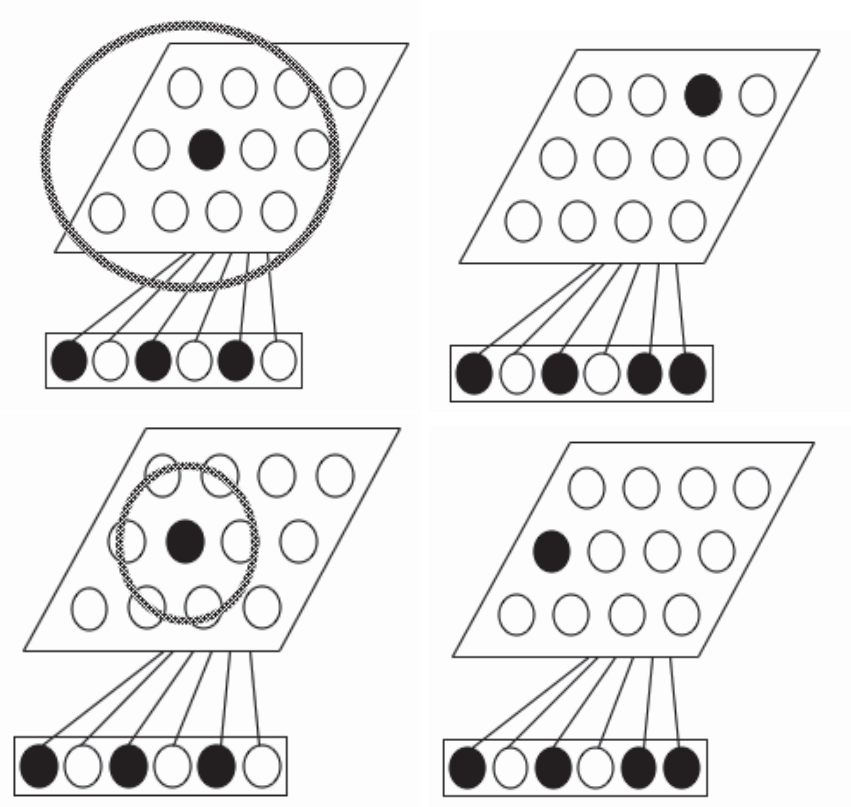

FIGURE 14.9 *Top:* Initially, similar input vectors excite neurons that are far apart, so that the neighbourhood (shown as a circle) needs to be large. *Bottom:* Later on during training the neighbourhood can be smaller, because similar input vectors excite neurons that are close together.

```
# Set up the map distance matrix
mapDist = np.zeros((self.x*self.y,self.x*self.y))
for i in range(self.x*self.y):
    for j in range(i+1,self.x*self.y):
        mapDist[i,j] = np.sqrt((self.map[0,i] - self.map[0,j])**2 + (self.map[1,i] - self.map[1,j])**2)
 mapDist[j,i] = mapDist[i,j]

# Within the loop, select the neighbours
# Find the neighbours and update their weights
neighbours = np.where(mapDist[best[i]]<=self.nSize,1,0)
neighbours[best[i]] = 0
self.weights += self.eta_n * neighbours*np.transpose((inputs[i,:] - np.transpose(self.weights)))
```

There is another way to initialise the weights in the network, which is to use Principal Components Analysis (which is described in Section 6.2) to find the two (assuming that the map is two-dimensional) largest directions of variation in the data and to initialise the weights so that they increase along these two directions:

```
dummy1,dummy2,evals,evecs = pca.pca(inputs,2)
self.weights = np.zeros((self.nDim,x*y))
for i in range(x*y):
    for j in range(self.mapDim):
        self.weights[:,i] += (self.map[j,i]-0.5)*2*evecs[:,j]
```

This means that the ordering part of the training has already been done in the initialisation, and so the algorithm can be trained with small neighbourhood size from the start. Obviously, this is only possible if the training of the algorithm is in batch mode, so that you have all of the data available for training right from the start. This should be true for the SOM anyway—it is not designed for on-line learning. This can be a bit of a limitation, because there are many cases where we would like to do unsupervised on-line learning.

There are a couple of different things that we can do. One is to ignore that constraint and use the SOM anyway. This is fairly common. However, the size of the map really starts to matter, and there is no guarantee that the SOM will converge to a solution unless batch learning is applied. The alternative is to use one of a variety of networks that are designed to deal with exactly this situation. There are a fair number of these, but Fritzke's "Growing Neural Gas" and Marsland's "Grow When Required" Network are two of the more common ones.

14.3.3 Self-Organisation

You might be wondering what the self-organisation in the name of the SOM is. A particularly interesting aspect of feature mapping is that we get a global ordering of the neurons in the network, despite the fact that the interactions are all local, since neurons that are very far apart do not interact with each other. We thus get a global ordering of the space using

only a set of local interactions, which is amazing. This is known as self-organisation, and it appears everywhere. It is part of the growing science of complexity. To see how common self-organisation is, consider a flock of birds flying in formation. The birds cannot possibly know exactly where each other are, so how do they keep in formation? In fact, simulations have shown that if each bird just tries to stay diagonally behind the bird to its right, and fly at the same speed, then they form perfect flocks, no matter how they start off and what objects are placed in their way. So the global ordering of the whole flock can arise from the local interactions of each bird looking to the one on its right (or left).

14.3.4 Network Dimensionality and Boundary Conditions

We typically think about applying the SOM algorithm to a 2D rectangular array of neurons (as shown in Figure 14.6), but there is nothing in the algorithm to force this. There are cases where a line of neurons (1D) works better, or where three dimensions are needed. It depends on the dimensionality of the inputs (actually on the intrinsic dimensionality, the number of dimensions that you actually need to represent the data), not the number that it is embedded in. As an example, consider a set of inputs spread through the room you are in, but all on the plane that connects the bottom of the wall to your left with the top of the wall to your right. These points have intrinsic dimensionality two since they are all on the plane, but they are embedded in your three-dimensional room. Noise and other inaccuracies in data often lead to it being represented in more dimensions than are actually required, and so finding the intrinsic dimensionality can help to reduce the noise.

We also need to consider the boundaries of the network. In some cases, it makes sense that the edges of the map of neurons is strictly defined — for example, if we are arranging sounds from low pitch to high pitch, then the lowest and highest pitches we can hear are obvious endpoints. However, it is not always the case that such boundaries are clearly defined. In this case we might want to remove the boundary conditions. We can do this by removing the boundary by tying the ends together. In 1D this means that we turn a line into a circle, while in 2D we turn a rectangle into a torus. To see this, try taking a piece of paper and bend it so that the top and bottom edges line up. You've now got a tube. If you bend the tube round so that the two open ends meet up you have a circle of tube known as a torus. Pictures of these effects are shown in Figure 14.10. In effect, it means that there are no neurons on the edge of the feature map. The choice of the number of dimensions and the boundary conditions depends on the problem that we are considering, but it is usually the case that the torus works better than the rectangle, although it is not always clear why.

The one cost that this has is that the map distances get more complicated to calculate, since we now need to calculate the distances allowing for the wrap around. This can be done using modulo arithmetic, but it is easier to think about taking copies of the map and putting them around the map, so that the original map has copies of itself all around: one above, one below, to the right and left, and also diagonally above and below, as is shown in Figure 14.11. Now we keep one of the points in the original map, and the distance to the second node is the smallest of the distances between the first node and the copies of the second node in the different maps (including the original). By treating the distances in x and y separately, the number of distances that has to be computed can be reduced.

As with the competitive learning algorithm that we considered earlier, the size of the SOM is defined before we start learning. The size of the network (that is, the number of neurons that we put into it) decides how fine-grained the learning is. If there are very few neurons, then the best that the network can do is to find gross generalisations that link the data. However, if there are very large numbers of neurons, then the network can

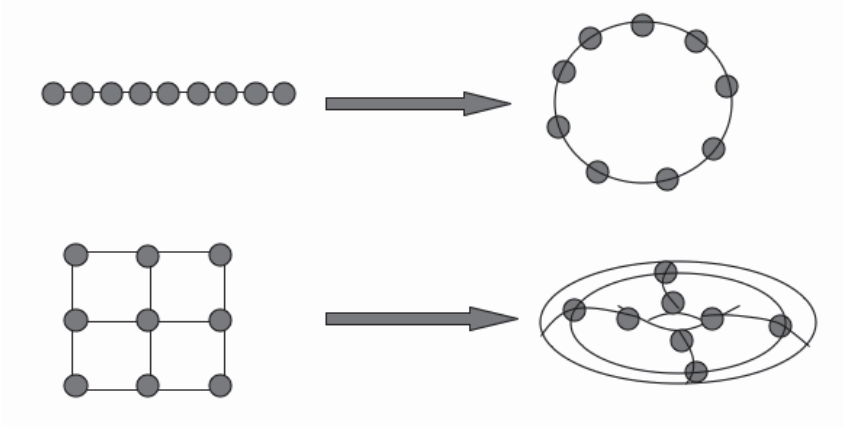

FIGURE 14.10 Using circular boundary conditions in 1D turns a line into a circle, while in 2D it turns a rectangle into a torus.

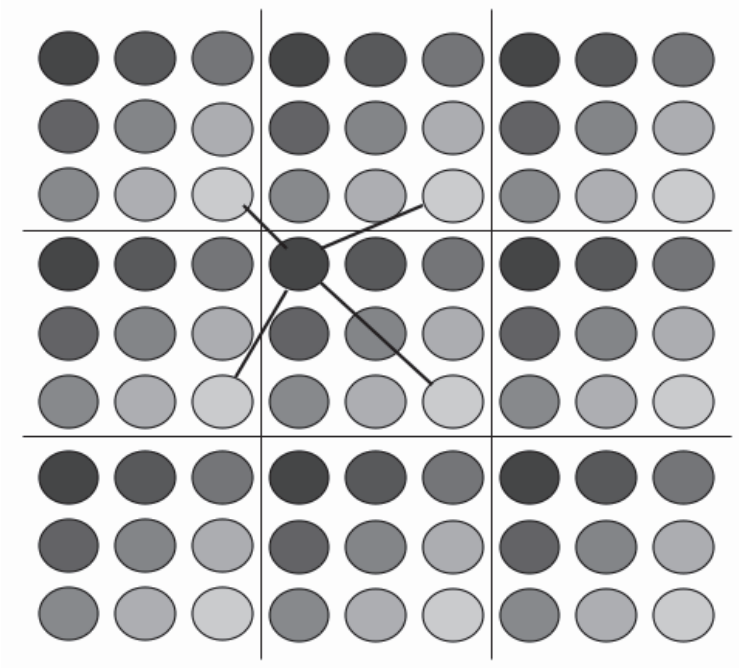

FIGURE 14.11 One way to compute distances between points without any boundary on the map is to imagine copies of the entire map being placed around the original, and picking the shortest of the distances between a node and any of the copies of the other node.

represent every input without ever needing to generalise at all. This is yet another example of overfitting. Clearly, then, choosing the correct size of network is important. The common approach is to test out several different sizes of network, such as 5×5 and 10×10 and see how well the network learns.

14.3.5 Examples of Using the SOM

As a first example of using the SOM, and one that shows the topological ordering of the network, consider training the network on a set of two-dimensional data drawn at random from a uniform distribution in $[-1, 1]$ in both directions. If the network weights are started off randomly, then initially the network is completely disordered (as shown in the top-left picture in Figure 14.12), but after 10 iterations of training the network is ordered so that neighbouring nodes map to data that is close together (bottom-left). Using PCA to initialise the map is not especially useful for this dataset, but it does speed things up: only five iterations through the dataset produce the output shown on the bottom-right of the figure, where it started from the version on the top-right.

For two examples of using the SOM on non-random data, where we can expect to see some actual learning, we will first look at the iris data that we used with the k-means algorithm earlier in this chapter. Figure 14.13 shows a plot of which node of a 5×5 Self-Organising Map was the best match on a set of test data after training for 100 iterations. The three different classes are shown as different shapes (squares, plus triangles pointing up and down), but remember that the network did not receive any information about these target classes. It can be seen that the examples in each of the three classes form different clusters in the map. Looking at the figure, you might be wondering if it is possible to use the plot to identify the different classes by assuming that they are separated in the map. This has been investigated—often by using methods similar to those of Linear Discriminant Analysis that are described in Section 6.1—with some success, and a reference is provided at the end of the chapter.

A more difficult problem is shown in Figure 14.14. The data are the `ecoli` dataset from the UCI Machine Learning repository, and the class is the localisation site of the protein, based on a set of protein measurements. The results with this dataset when testing are not as clearly impressive (but note that the MLP gets about 50% accuracy on this dataset, and that has the target data, which the SOM doesn't). However, the clusters can still be seen to some extent, and they are very clear in the training data. Note that the boundary conditions can make things a little more complicated, since the cluster does not necessarily respect the edges of the map.

FURTHER READING

There is a book by Kohonen, the inventor of the SOM, that provides a very good overview (if rather dated, now) of the area:

- T. Kohonen. *Self-Organisation and Associative Memory*, 3rd edition, Springer, Berlin, Germany, 1989.

The two on-line self-organising networks that were mentioned in the chapter were:

- B. Fritzke. A growing neural gas network learns topologies. In Gerald Tesauro, David S. Touretzky, and Todd K. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, MIT Press, Cambridge, MA, USA, 1995.

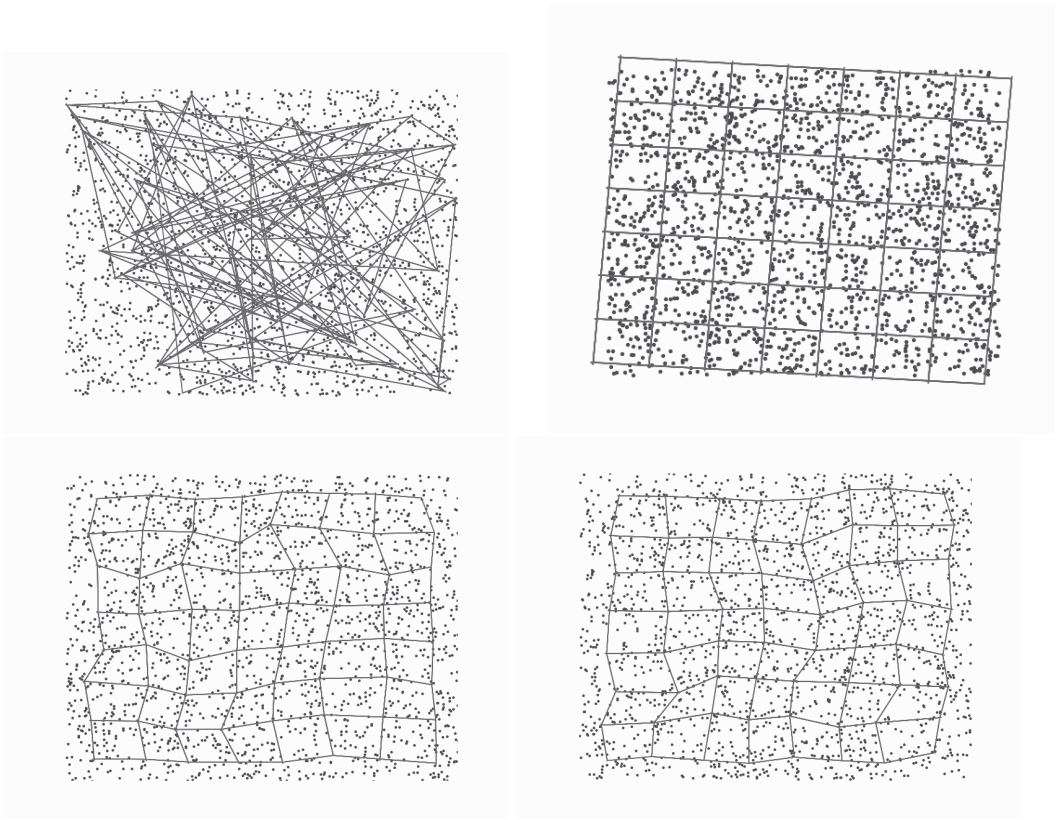

FIGURE 14.12 Training the SOM on a set of uniformly randomly sampled two-dimensional data in the range $[-1, 1]$ in both dimensions. *Top:* Initialisation of the map using *left:* random weights and *right:* PCA (the randomness in the data means that the directions of variation are not necessarily along the obvious directions). *Bottom:* The output after just 10 iterations of training on the left, and 5 on the right, both with typical parameter values.

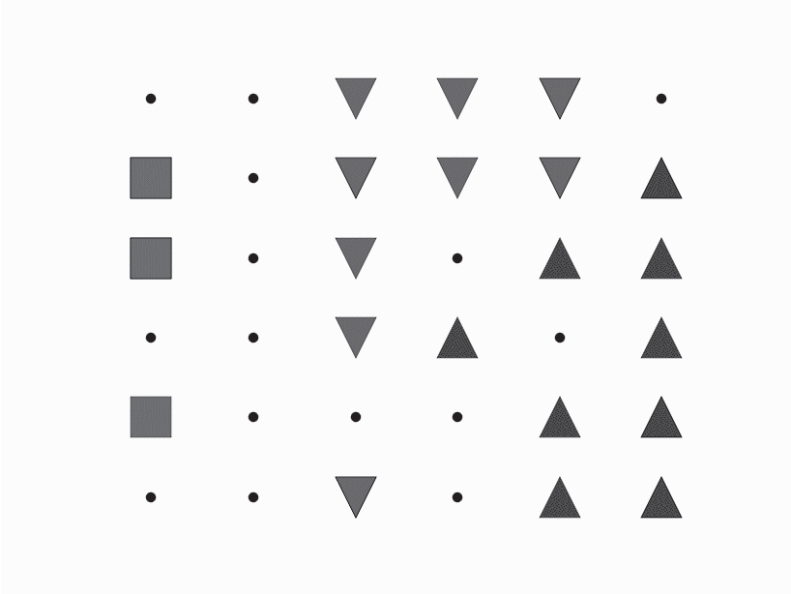

FIGURE 14.13 Plot showing which nodes are the best match according to class, with the three shapes corresponding to three different classes in the iris dataset. The small dots represent nodes that did not fire.

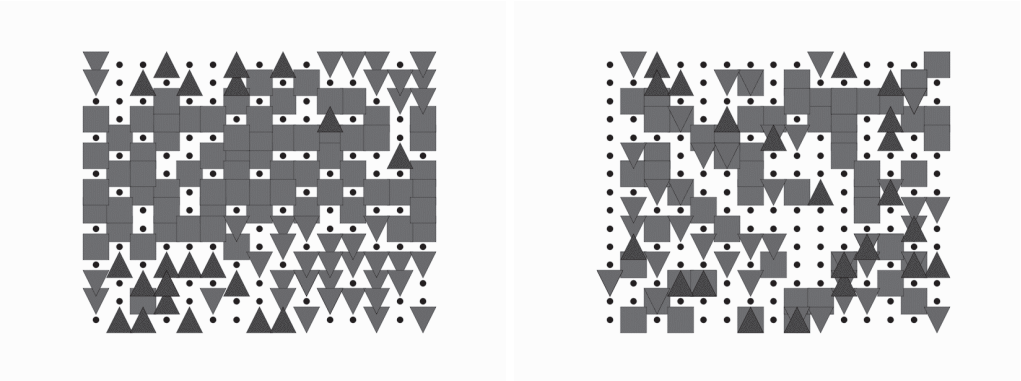

FIGURE 14.14 Plots showing which nodes are the best match according to class, with the three shapes corresponding to three different classes in the *E. coli* dataset, tested on *left:* the training set and *right:* a separate test set. The small dots represent nodes that did not fire.

- S. Marsland, J.S. Shapiro, and U. Nehmzow. A self-organising network that grows when required. *Neural Networks*, 15(8-9):1041–1058, 2002.

A possible reference on processing the data in the map in order to identify clusters is:

- S. Wu and T.W.S. Chow. Self-organizing-map based clustering using a local clustering validity index. *Neural Processing Letters*, 17(3):253–271, 2003.

Books that cover the area include:

- Section 10.14 of R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*, 2nd edition, Wiley-Interscience, New York, USA, 2001.
- Chapter 9 of S. Haykin. *Neural Networks: A Comprehensive Foundation*, 2nd edition, Prentice-Hall, New Jersey, USA, 1999.
- Section 9.3 of B.D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, Cambridge, UK, 1996.

PRACTICE QUESTIONS

Problem 14.1 What is the purpose of the neighbourhood function in the SOM? How does it change the learning?

Problem 14.2 A simplistic intruder detection system for a computer network consists of an attempt to categorise users according to (i) the time of day they log in, (ii) the length of time they log in for, (iii) the types of programs they run while logged in, (iv) the number of programs they run while logged in. Suggest how you would train a SOM and the naïve Bayes' classifier to perform the categorisation. What preprocessing of the data would you do, how much data would you need, and how large would you make the SOM? Do you think that such a system would work for intruder detection?

Problem 14.3 The Music Genome Project (`http://www.pandora.com`) does not work by using a SOM. But it could. Describe how you would implement it.

Problem 14.4 A bank wants to detect fraudulent credit card transactions. They have data for lots and lots of transactions (each transaction is an amount of money, a shop, and the time and date) and some information about when credit cards were stolen, and the transactions that were performed on the stolen card. Describe how you could use a competitive learning method to cluster people's transactions together to identify patterns, so that stolen cards can be detected as changes in pattern. How well do you think this would work? There is much more data of transactions when cards are not stolen, compared to stolen transactions. How does this affect the learning, and what can you do about it?

Problem 14.5 It is possible to use any competitive learning method to position the basis functions of a Radial Basis Function network. The example code used k-means. Modify it to use the SOM instead and compare the results on the `wine` and `yeast` datasets.

Problem 14.6 For the `wine` dataset, experiment with different sizes of map, and boundary conditions. How much difference does it make? Can you use the principal components in order to set the size automatically?