actually fires or not, based on a random sample. The probability computation for a neuron is something like:

```
sumin = self.visiblebias + np.dot(hidden,self.weights.T)
self.visibleprob = 1./(1. + np.exp(-sumin))
```

and so the problem is to decide whether or not the neuron actually fires. The simplest way to do it is to sample a uniform random number between 0 and 1, and then check whether or not that is bigger. Two possible ways to do that are shown in the next code snippet:

```
self.visibleact1 = (self.visibleprob>np.random.rand(np.shape(self.↩
visibleprob)[0],self.nvisible)).astype('float')
self.visibleact2 =np.where(self.visibleprob>np.random.rand(self.visibleprob.↩
shape[0],self.visibleprob.shape[1]),1.,0.)
```

There is no difference between the output of these two lines, so at first glance there is nothing to choose between them. However, there is a difference. To see it we need to explore the tools that Python provides to time things, with the `TimeIt` module. This can be used in different ways, but as a demonstration, we will just use it at the Python command line. The TimeIt module runs a command (or set of commands) a specified number of times, and returns information about how long that took.

The next few lines show the results of running this on my computer, where the first argument to the `timeit` method is the command to time, the second is the setup to do (which is not included in the time), and the last one is the number of times to run it. So this code makes a fairly small array of random numbers and uses the two approaches to turn them into binary firing values, performing it 1,000 times.

```
>>> import timeit
>>> timeit.timeit("h = (probs > np.random.rand(probs.shape[0],probs.shape[1])↩
).astype('float')",setup="import numpy as np; probs =
np.random.rand(1000,↩
100)",number=1000)
2.2446439266204834
>>> timeit.timeit("h= np.where(probs>np.random.rand(probs.shape[0],probs.↩
shape[1]),1.,0.)",setup="import numpy as np; probs =
np.random.rand(1000,100)↩
",number=1000)
5.140886068344116
```

It can be seen that the second method takes more than twice as long as the first method, although neither of them is that fast. Since this is a computation that will be performed many, many times when the RBM runs, it is definitely worth using the first version rather than the second.

You might be wondering why the truth values are cast as floats rather than ints. We have seen the reason for this type of casting before which is that NumPy tends to cast things as the lowest complexity type, and so if the activations are cast as integers, then the whole calculation of the probabilities that are based on this input at the next layer can also be cast as integers, which obviously causes large errors.

Using this code to compute the activations of the two layers, the steps for simple contrastive divergence in Python match the algorithm description very clearly:

```
def contrastive_divergence(self,inputs,labels=None,dw=None,dwl=None,⤸
dwvb=None,dwhb=None,dwlb=None,silent=False):
    # Clamp input into visible nodes
    visible = inputs
    self.labelact = labels

    for epoch in range(self.nepochs):
 # Awake Phase
 # Sample the hidden variables
 self.compute_hidden(visible,labels)

# Compute <vh>_0
 positive = np.dot(inputs.T,self.hiddenact)
 positivevb = inputs.sum(axis=0)
 positivehb = self.hiddenprob.sum(axis=0)

# Asleep Phase
 # Do limited Gibbs sampling to sample from the hidden distribution
 for j in range(self.nCDsteps):
     self.compute_visible(self.hiddenact)
     self.compute_hidden(self.visibleact,self.labelact)

# Compute <vh>_n
 negative = np.dot(self.visibleact.T,self.hiddenact)
 negativevb = self.visibleact.sum(axis=0)
 negativehb = self.hiddenprob.sum(axis=0)

# Learning rule (with momentum)
 dw = self.eta * ((positive - negative) / np.shape(inputs)[0] - self.⤸
 decay*self.weights) + self.momentum*dw
self.weights += dw
dwvb = self.eta * (positivevb - negativevb) / np.shape(inputs)[0] + self.⤸
 momentum*dwvb
self.visiblebias += dwvb
dwhb = self.eta * (positivehb - negativehb) / np.shape(inputs)[0] + self.⤸
 momentum*dwhb
self.hiddenbias += dwhb
error = np.sum((inputs - self.visibleact)**2)

visible = inputs
```

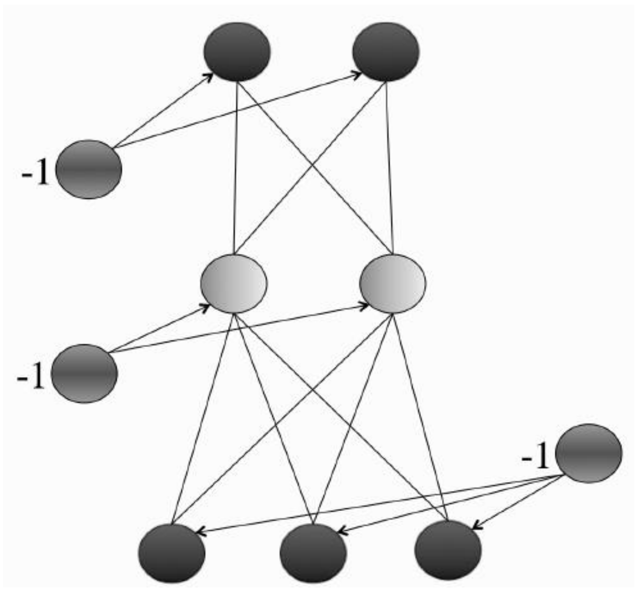

FIGURE 17.8 An RBM for supervised learning, with an extra layer of 'label' nodes, also connected with symmetric weights. However, these nodes use soft-max activation instead of logistic activation.

```
self.labelact = labels
```

17.2.3 Supervised Learning

The RBM performs pattern completion, just like the Hopfield network, so that after training, when a corrupted or partial input is shown to the network (by putting them into the visible units) the network relaxes to a lower energy trained state. However, it is possible to extend the RBM so that it performs classification. This is done by adding an additional layer of soft-max units (for a reminder about soft-max, see Section 4.2.3) that also have symmetric weights (see Figure 17.8).

The activation of these nodes is soft-max rather than logistic, so that there is only one neuron (one class) activated for each input. The training rule for this extra set of weights (and the corresponding bias weights) is also based on contrastive divergence.

The additions that this makes to an implementation of the RBM are fairly simple, with the most obvious difference in the sampling of the visible nodes, since there are now two sets of them: the input nodes and the label nodes ($\mathbf{l}$), so we are computing $p(\mathbf{v}|\mathbf{h}, \mathbf{W})$ and $p(\mathbf{l}|\mathbf{h}, \mathbf{W}')$, where $\mathbf{W}'$ are the extra weights connecting the hidden nodes to the label nodes; these weights are not included in the conditioning of each other since they are independent given the hidden nodes. However, the probabilities of the hidden nodes are conditioned on both of them (and both sets of weights): $p(\mathbf{h}|\mathbf{v}, \mathbf{l}, \mathbf{W}, \mathbf{W}')$.

Since the label nodes are soft-max units, the activation of them is different, as was described in Section 4.2.3. However, the implementation can be a little awkward in that if we use the soft-max equation as it stands, then the numbers get very large (since we are calculating e^x for quite large values of x). One simple solution to this is to subtract off the largest value so that the numbers are all 0 or below. This does make for quite complicated code, unfortunately:

```
# Compute label activations (softmax)
if self.nlabels is not None:
 sumin = self.labelbias + np.dot(hidden,self.labelweights.T)
 summax = sumin.max(axis=1)
 summax = np.reshape(summax,summax.shape+(1,)).repeat(np.shape(sumin)[1],↩
 axis=-1)
 sumin -= summax
normalisers = np.exp(sumin).sum(axis=1)
 normalisers = np.reshape(normalisers,normalisers.shape+(1,)).repeat(np.↩
 shape(sumin)[1],axis=-1)
 self.labelact = np.exp(sumin)/normalisers
```

Figure 17.9 shows the outputs of a RBM with 50 hidden nodes learning about 3 letters (‘A’, ‘B’, and ‘S’) from the Binary Alphadigits dataset without knowing about the labels, while Figure 17.10 shows the same examples, but with a labelled RBM. The algorithm had 1,000 epochs of learning. The top row shows the training set (with 20 examples of each letter) and the reconstructed version of each member of the training set, while the bottom row shows the test set of the remaining 57 examples (19 of each) and their reconstructions. It can be seen that the reconstructions are mostly pretty good. The labelled algorithm got 0 training examples wrong, and 5 of the test examples wrong, whereas the unlabelled version got 0 training examples wrong, but 7 test examples wrong.

17.2.4 The RBM as a Directed Belief Network

We have just seen that the RBM does a pretty good job of learning about a fairly complex dataset, and that the fact that we can use the network as a generative model helps us to see what it is learning. One way to understand the power of the RBM is to see that it is equivalent to a directed network that has an infinite number of layers, all consisting of the same stochastic neurons as the RBM, and with the same weight matrix connecting each pair of sequential layers, as is shown in Figure 17.11.

For a given set of weights we can use this network to generate possible inputs that were used in training by putting a random configuration into the nodes of an infinitely deep layer. The activations of the previous layer can then be inferred using the weights, and choosing the binary state of each neuron from a probability distribution based on the parents. This procedure can be iterated until the visible nodes are reached, and this gives us a reconstruction of a possible input.

It is also possible to go ‘up’ the network, from the visible nodes through the layers, by using the transpose of the weight matrix that is used for going from top to bottom. This enables us to infer the factorial distribution (that is, just the product of independent distributions) over a hidden layer by setting values for the visible layer below it. This is a factorial distribution since the nodes in the hidden layer are independent of each other. We can follow this process up the layers, and it enables us to sample from the posterior distribution over the hidden layers.

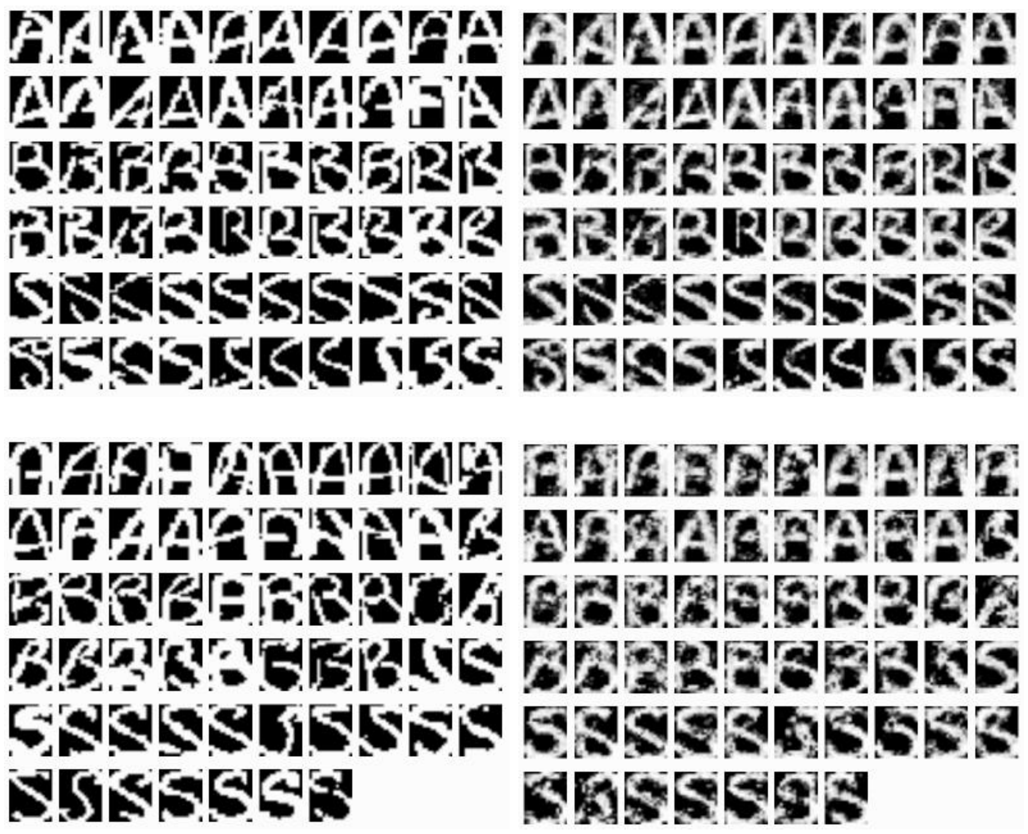

FIGURE 17.9 Training with an unlabelled RBM. *Top left:* Training set, *top right:* reconstructed versions of the training set, *bottom left:* Test set, *bottom right:* reconstructed versions of the test set. A few errors can be seen in this last set, such as the last picture on the 5th line, where an ‘S’ has been reconstructed as a ‘B’.

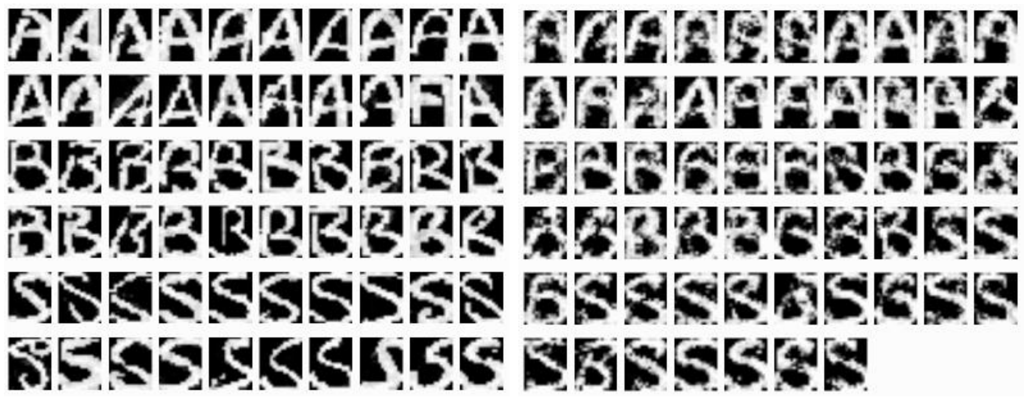

FIGURE 17.10 Training with a labelled RBM. *Left:* reconstructed versions of the training set, *right:* reconstructed versions of the test set.

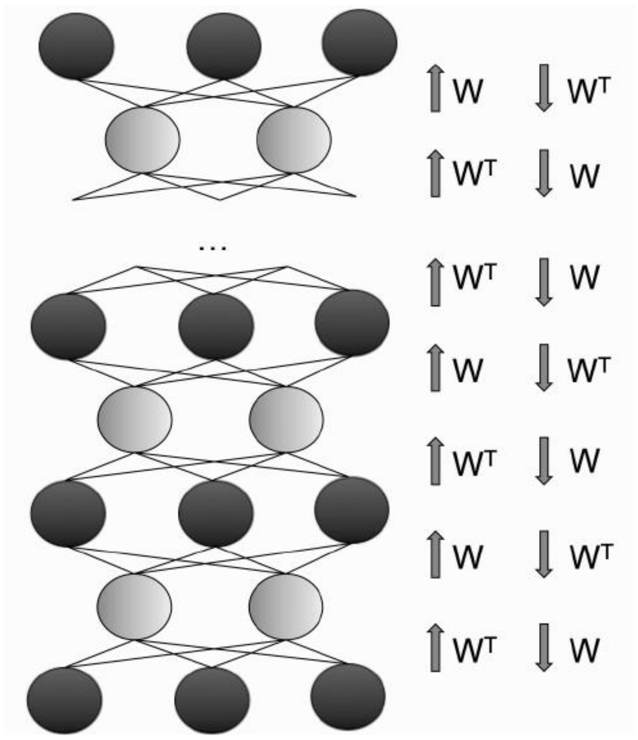

FIGURE 17.11 A Directed Belief Network with infinite layers.

Since we can sample from the posterior distribution, we can also compute derivatives of the probability of the data (and more pertinently, the log probability of the data):

$$\frac{\partial \log p(\mathbf{v}^0)}{\partial w_{ij}} = \langle h_j^0(v_i^0 - \hat{v}_i^0)\rangle, \tag{17.40}$$

where $\mathbf{v}^0$ denotes the vector of values in visible layer 0, and $\hat{v}_i$ is the probability that neuron i is firing if the visible vector is reconstructed from the sampled hidden values. Since the weight matrix is the same between each pair of layers, $\hat{v}_i^0 = v_i^1$.

Since we have the same weight matrix at each layer, the full derivative of the weight needs to have the contribution of all of the layers included in it:

$$\frac{\partial \log p(\mathbf{v}^0)}{\partial w_{ij}} = \langle h_j^0(v_i^0 - v_i^1)\rangle + \langle v_i^1(h_j^0 - h_j^1)\rangle + \langle h_j^1(v_i^1 - v_i^2)\rangle + \langle v_i^2(h_j^1 - h_j^2)\rangle + \ldots \tag{17.41}$$

If you look at this carefully, you will notice that most of the terms cancel out, and the final result is:

$$\frac{\partial \log p(\mathbf{v}^0)}{\partial w_{ij}} = \langle v_i^0 h_j^0\rangle - \langle v_i^\infty h_j^\infty\rangle, \tag{17.42}$$

which is exactly the same as the full Boltzmann machine. Since the two networks minimise the same gradient and are based on the same sampling steps, they are equivalent.

This actually lets us see something important, which is the RBM seems to cancel out some of the need for explaining away when performing inference. Explaining away is one of the main challenges with performing inference in a belief network. We want to be able to choose between possible competing reasons for something happening, so that if we see that a visible node is on, and there are two competing hidden nodes that could have caused it, they don't both get switched on automatically.

To see why explaining away matters, suppose that you are sitting a multiple-choice exam for a topic that you know nothing about (hopefully not machine learning any more!). If you were to get a passing grade for that exam, then the possible reasons could be that you got very lucky with your guesses, or that the examiner mixed up your exam paper with somebody who did understand the material.

Both of these reasons are quite unlikely, and they are independent of each other. So if you see the class genius looking shocked when she collects her result, then you can deduce that your exam papers have been swapped over, and stop thinking about buying a lottery ticket on the way home. In other words, although the two explanations are independent of each other, they are made conditionally dependent by the fact that they are both explanations for your exam success.

Explaining away was not a problem in the infinite belief network that we have just seen, but it would be for a finite one. Consider a single hidden layer, with a data vector clamped on the visible inputs. The nodes in the hidden layer would have a distribution that includes conditional dependencies between them, based on the data vector. In the infinite version this doesn't happen, since the other hidden layers cancel them out. These cancelling-out terms are sometimes known as complementary priors.

The RBM is of interest in itself, but it is what you can build it up into that is even more interesting, and this fact of equivalence with the infinite directed belief network will help with the required algorithm, as we shall see in the next section.

17.3 DEEP LEARNING

We saw in Chapter 4 that an MLP can learn any arbitrary decision surface. However, the fact that it can do it only tells part of the story of learning. It might very well need lots and lots of nodes in order to learn some interesting function, and that means even more weights, and even more training data, and even more learning time, and even more local minima to get stuck in. So while it is true in theory, it doesn't mean that the MLP is the last word in learning, not even supervised learning.

One way to view all of the algorithms that we have talked about in this book is that they are *shallow* in the sense that they take a set of inputs and produce (linear or nonlinear) combinations of those inputs, but nothing more than that. Even the methods that are based on trees only consider one input at a time, and so are effectively just a weighted combination of the inputs.

This doesn't seem to be the way that the brain works, since it has columns built up of several layers of neurons, and it also isn't the way that we seem to do things like analysing images. For example, Figure 17.12 shows an image and different representations and sets of features that can be derived from the image. If we want to perform image recognition then we can't just feed in the pixel values into our machine learning algorithm, since all that this will do is find combinations of these pixel values and try to perform classification based on that. Instead, we derive sets of features of interest from the images, and feed those into the learning algorithm. We choose the features that seem to be useful according to our knowledge of the problem that we wish to solve and the appearance of the images; for example, if we are looking at the shape of objects, then edges are more useful than textures. It might also be useful to know how circular objects are, etc. Further, if the intensity of lighting in the images change, so that some are dark and some are light, then pixel intensities are not very useful, but edges and other features based on derivatives of the image intensity might well be.

All of this knowledge is put into the choice of the inputs features by the human investigator, and then the derived features are fed into the machine learning algorithm, with the hope that combinations of these derived features will be enough to enable the recognition.

However, when we look at an image we appear to perform several different recognition problems, looking at shape, texture, colour, etc., both independently and together. It seems that we split the recognition problem up into lots of sub-problems, and solve those and combine the results of those sub-problems to make our decision. We go from the colour of individual parts of the image (sort-of like pixels) through progressively more abstract representations until we combine them all to recognise the whole image.

We can make deep networks, where there are progressively more combinations of derived features from the inputs with the MLP, since we can just add more layers and then use the back-propagation algorithm to update the weights. But the search space that the algorithm is trying to find a minimum of gets massively larger as we do this, and the estimates of the gradients in that space that the back-propagation algorithm is making get noisier and noisier. So making deep networks isn't so hard, but training them is. In fact, they should actually need less training overall than a shallow network that has the equivalent expressivity, for reasons that I won't go into here, but that doesn't make them easier to train.

The first experiments that people performed into deep learning were mostly with autoencoders, which we saw in Section 4.4.5. These were MLPs where the inputs and outputs were clamped together, so that the (smaller number of) hidden nodes produced a lower dimensional representation of the inputs. We also saw that this could be used to perform pattern completion.

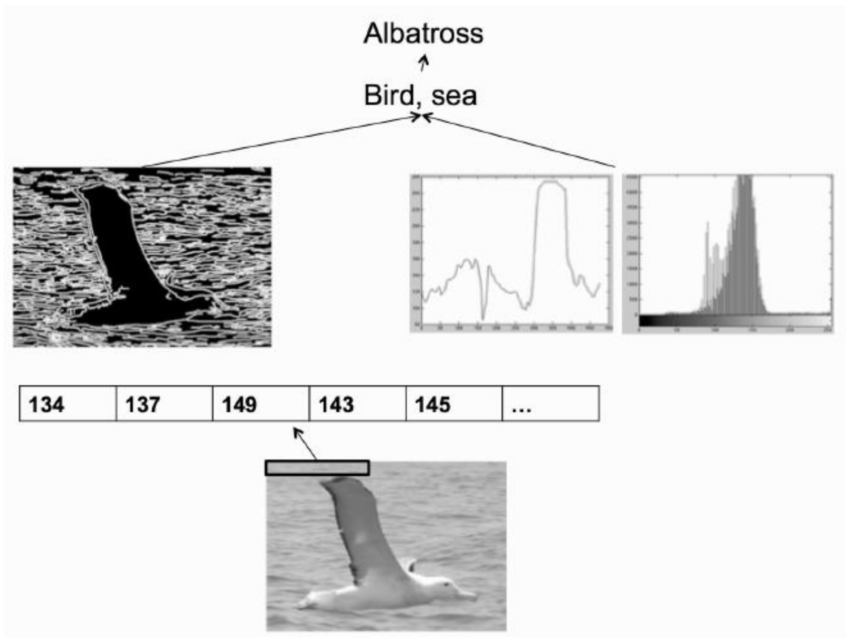

FIGURE 17.12 The idea of deep learning is that initial analysis of an image such as the bird at the bottom can only deal with the pixel values. The learners at different levels can produce higher-order correlations of the data, so that eventually the whole system can learn more complicated functions.

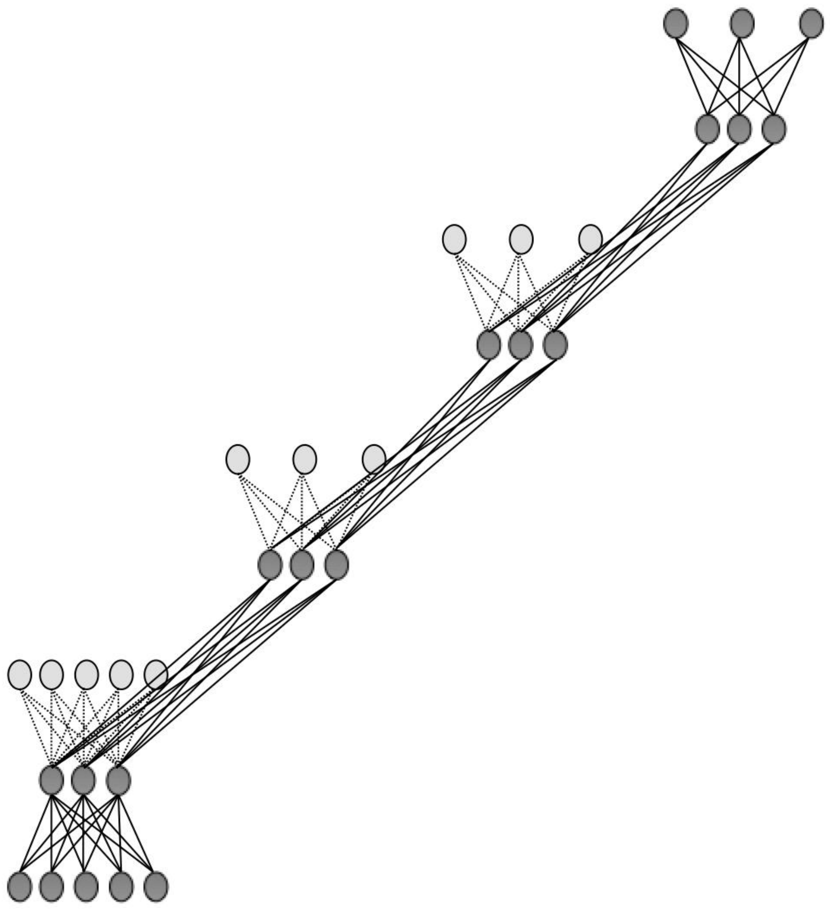

FIGURE 17.13 A schematic of a deep network built up of a set of autoencoders. The hidden layer of one autoencoder, which provides a representation of the input data it sees, is used as the input to the next one. The second half of each autoencoder, which is the reconstruction of the inputs, is shown in lighter grey and with dotted weight lines.

The idea is fairly simple: we train a single autoencoder, and then use the hidden layer of that network as the input to another one. This second network learns a higher-order representation of the initial inputs, that are based on the activations of the hidden nodes of the original network. Progressively more autoencoders can be trained and stacked on top of one another, and if the purpose is to do classification or regression, then a Perceptron can be added at the top, to take the activations of the final set of hidden nodes and perform supervised learning on them. Figure 17.13 shows a schematic of this kind of learning architecture.

The simplicity of the scheme is also the problem with it. Each autoencoder is trained in a pretty much unsupervised way; it is normal back-propagation learning, but the input and the desired output are the same, so there is no real information about what the hidden layer should be learning. There is some real supervised learning in the Perceptron at the final layer of the network, but this network has to deal with what it gets as the inputs, there is no error signal that informs the weights in all of the lower-down autoencoders and enables them to be trained further to produce more useful representations of the inputs.

There is some similarity between autoencoders and RBMs; they perform the same job, but the autoencoder is directional in that the weights run from input to hidden node to output, while the RBM weights are symmetrical. This means that we can run the RBM backwards: we can take samples of the hidden nodes and infer the values of the visible nodes that gave rise to them. This is a generative model of the type of inputs that the network sees. And this means that we can get information from the top of a stack of RBMs and push it back down through the RBMs all the way back to the input visible units, which gives us a chance of actually changing the weights of these RBMs, and so the representations that they find.

Deep learning is a very popular area for research at the moment, and various companies like Google obviously believe it is important, since they are employing a large number of deep learning researchers, and buying up companies that have been successful in developing applications based upon these ideas.

A set of stacked RBMs is known as a Deep Belief Network (DBN) and it is the topic of the next section.

17.3.1 Deep Belief Networks (DBN)

Conceptually, the DBN is pretty much the same thing that we have just seen with autoencoders. It consists of a series of unlabelled RBMs stacked together, with a labelled RBM at the very top. However, while creating the architecture is simple, we need to do some work to work out how to train it.

The first hope would be that we set up our stack of RBMs, and then use CD learning to train all of the weights in the network at the same time. However, it turns out that it takes a very long time for the network to settle to a distribution, and this involves lots of sampling steps up and down the network.

Instead, we will start greedily, by sequentially training the series of RBMs, just as we discussed doing for the autoencoders above. We clamp an input onto the visible nodes, and train this RBM, which will learn a set of symmetric weights that describe a generative model of the inputs. We then sample the hidden nodes of the RBM, and sequentially train a series of these RBMs, each unlabelled, with the visible nodes for layer i being clamped to samples of the hidden nodes for layer $i-1$. At the top layer we use an RBM with labels, and train that. This is the complete greedy learning algorithm for the DBN, and it is exactly the same as we could do with an autoencoder. However, we haven't finished yet.

At this stage we can recognise that there are two purposes to the network: to recognise inputs (that is, to do normal classification) based on the visible nodes and to generate samples that look like the inputs based on the hidden nodes. The RBM is fully symmetric and so the same weight matrix is used for these two purposes. However, once we start to add extra layers of RBMs above and below, with success training of each RBM based on the training of the one below it, things get a bit out of sync between working upwards from the visible nodes at the bottom to perform recognition, and working downwards from the output nodes to generate samples, principally because in the generative model the weights were set assuming that the hidden node probabilities came from the recognition model, and this is no longer true, since they come from the layers above the current one, and therefore will be different to the values that they were trained on.

For these reasons, there are two parts to the training. In the first, greedy, part there is only one set of weights, and so the training is precisely that of a set of autoencoders, while in the second part the recognition weights and generative weights are decoupled, except for the labelled RBM at the top layer (since there the weights are still actually the same since there are no layers above to confuse things). We can now use a variant of the wake-sleep algorithm that we saw earlier. Starting at the visible layer with a clamped input, we use the recognition weights to pick a state for each hidden variable, and then adjust the generative weights using Equation (17.40). Once we have reached the associative memory RBM at the top, we train this normally and then create samples of the hidden units using (truncated) Gibbs sampling. These are then used to sample the visible nodes at the top-most unlabelled RBM, and a similar update rule (with the role of hidden and visible nodes switched) is used to train the recognition weights.

So initially there is only one set of weights, and these are cloned at the appropriate point in the learning, and then modified from there. The following code snippet shows one way to implement this in NumPy.

```
def updown(self,inputs,labels):

    N = np.shape(inputs)[0]

    # Need to untie the weights
    for i in range(self.nRBMs):
 self.layers[i].rec = self.layers[i].weights.copy()
 self.layers[i].gen = self.layers[i].weights.copy()

    old_error = np.iinfo('i').max
    error = old_error
    self.eta = 0
    for epoch in range(11):
 # Wake phase

v = inputs
for i in range(self.nRBMs):
     vold = v
     h,ph = self.compute_hidden(v,i)
     v,pv = self.compute_visible(h,i)

     # Train generative weights
     self.layers[i].gen += self.eta * np.dot((vold-pv).T,h)/N
     self.layers[i].visiblebias += self.eta * np.mean((vold-pv),axis=0)

     v=h

# Train the labelled RBM as normal
 self.layers[self.nRBMs].contrastive_divergence(v,labels,silent=True)

# Sample the labelled RBM
 for i in range(self.nCDsteps):
     h,ph = self.layers[self.nRBMs].compute_hidden(v,labels)
     v,pv,pl = self.layers[self.nRBMs].compute_visible(h)
```

```
# Compute the class error
 #print (pl.argmax(axis=1) != labels.argmax(axis=1)).sum()

# Sleep phase

# Initialise with the last sample from the labelled RBM
 h = v
for i in range(self.nRBMs-1,-1,-1):
      hold = h
      v, pv = self.compute_visible(h,i)
      h, ph = self.compute_hidden(v,i)

      # Train recognition weights
      self.layers[i].rec += self.eta * np.dot(v.T,(hold-ph))/N
      self.layers[i].hiddenbias += self.eta * np.mean((hold-ph),axis=0)

      h=v

old_error2 = old_error
old_error = error
error = np.sum((inputs - v)**2)/N
if (epoch%2==0):
           print epoch, error
if (old_error2 - old_error)<0.01 and (old_error-error)<0.01:
      break
```

This combination of a greedy and wake-sleep algorithm trains the RBN. It is possible to show that if the full maximum likelihood training (rather than CD learning) is used then this training regime will never reduce the log probability of the data under the generative model. However, in practice CD learning is always used since it gives good results in a reasonable time.

Both classification and generative modelling simply consist of choosing values for the appropriate nodes and then sampling your way up or down the set of layers of the network. The entire algorithm is given next.

The Deep Belief Network Algorithm

- **Initialisation**
  - create a set of unlabelled RBMs with pre-defined numbers of hidden nodes in each, and the corresponding number of visible nodes in the layer above; finish with a single labelled RBM.
  - initialise all weights to small (positive and negative) random values, usually with zero mean and 0.01 standard deviation
- **Greedy learning**
  - clamp the input vector on the visible units of the first RBM and train it using the CD learning algorithm in Section 17.2.1

- sample the hidden nodes of this RBM and use these values to set the visible units of the next RBM
- repeat up the stack until you reach the RBM with labels; train this one with supervised learning using the hidden layer of the topmost RBM as the input visible units and the labels as the output visible units
- **Wake-sleep**
  - for some pre-determined number of epochs, or until learning stops improving:
  - for each of the unlabelled RBMs (k):
    - create a copy of the weight matrix, separating those for recognition and for generation
    - set the visible nodes to the relevant inputs (the inputs or the hidden nodes of the network below)
    - sample the hidden nodes and use those samples to reconstruct the visible nodes
    - update the generative weights with:

      $$w_{ij}^{g,(k)} \leftarrow w_{ij}^{g,(k)} + \eta h_j (v_i - \hat{v}_i) \tag{17.43}$$

      where v_i is the input value of visible node i, $\hat{v}_i$ is the reconstructed version, and (k) indexes the RBMs
    - update the biases with:

      $$w_{\text{visible},ij} \leftarrow w_{\text{visible},ij} + \eta \text{mean}(v_i - \hat{v}_i) \tag{17.44}$$

  - train the labelled RBM as normal with CD learning
  - use alternating Gibbs' sampling for a small number of iterations to get samples for hidden and visible nodes of the labelled RBM
  - for each of the unlabelled RBMs (k), starting at the top:
    - initialise the hidden nodes with the samples from the visible nodes of the layer above
    - sample the visible nodes and use those samples to reconstruct the hidden nodes
    - update the recognition weights with:

      $$w_{ij}^{r,(k)} \leftarrow w_{ij}^{r,(k)} + \eta v_i (h_j - \hat{h}_j) \tag{17.45}$$

      where h_j is the input value of hidden node j and $\hat{h}_j$ is the reconstructed version
    - update the biases with:

      $$w_{\text{hidden},ij} \leftarrow w_{\text{hidden},ij} + \eta \text{mean}(h_j - \hat{h}_j) \tag{17.46}$$

There are no particular new surprises in the implementation of this pair of algorithms.

To see how well this works, we continue with the three characters from the Binary Alphadigits dataset that we used to demonstrate the RBM. Figure 17.14 shows the output for a DBN made up of three RBNs, each with 100 nodes in the hidden layer.

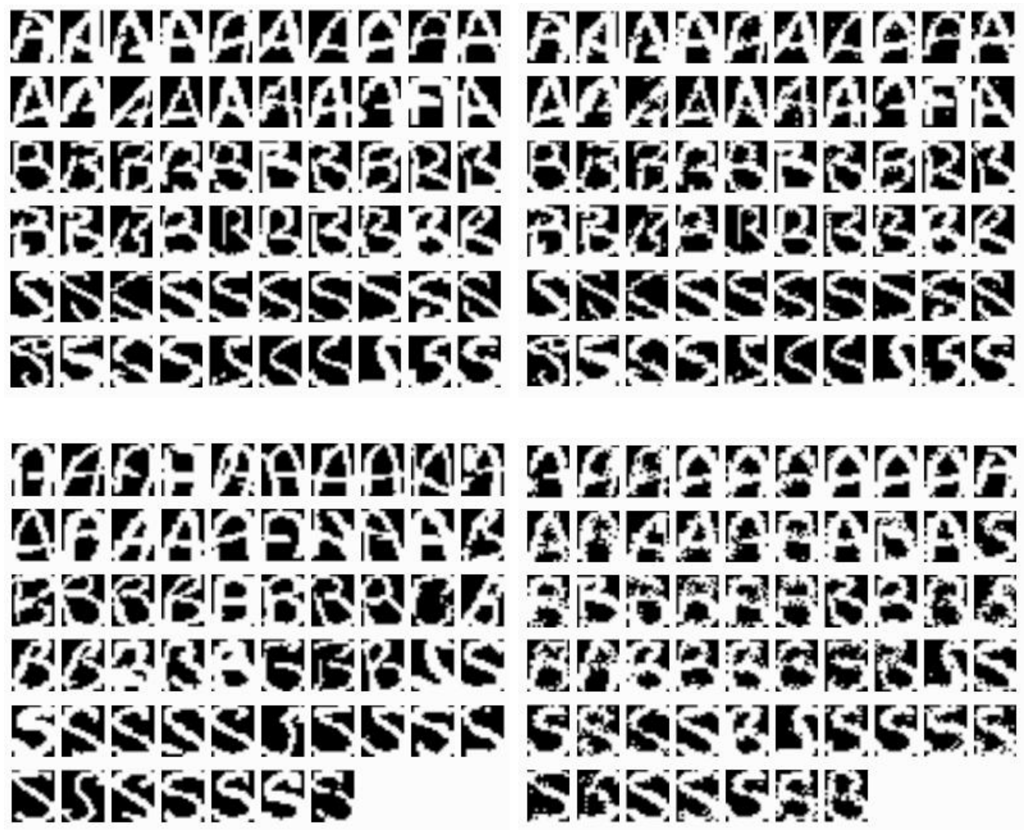

FIGURE 17.14 Training with the DBN. *Top left:* Training set, *top right:* reconstructed versions of the training set, *bottom left:* Test set, *bottom right:* reconstructed versions of the test set. The reconstructions are clearer than for the single RBN, although some errors are still visible such as the very last one.

FURTHER READING

The Hopfield network and some early work on Boltzmann Machines is covered in:

- D.J.C. MacKay. *Information Thoery, Inference and Learning Algorithms.* Cambridge University Press, Cambridge, UK, 2003.

For more up-to-date information, Hinton's papers are the best resource. If you are looking to make implementations of the RBM, then the following is definitely helpful:

- G. E. Hinton. A practical guide to training restricted Boltzmann machines. Technical Report UTML TR 2010-003, Department of Computer Science, University of Toronto, 2010.

For more on Deep Belief Networks, try:

- G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.
- Yoshua Bengio. Learning deep architectures for AI. Technical Report 1312, Dept. IRO, Universit'e de Montréal.
- Juergen Schmidhuber. Deep Learning in Neural Networks: An Overview `http://arxiv.org/abs/1404.7828`

PRACTICE QUESTIONS

Problem 17.1 In the Hopfield network the learning rate is not an important parameter. Work out why not, and justify the use of $\frac{1}{N}$.

Problem 17.2 Modify the Hopfield network code to make a continuous version of the network, using tanh() (which is available as `np.tanh()`) as the activation function. Produce a greyscale version of the digits problem and use it to recognise them.

Problem 17.3 The Travelling Salesman Problem (TSP) was discussed in Section 9.4. It can be solved using a Hopfield network by using an $N \times N$ network for N cities, with the columns representing the cities, and the rows representing the order in which they occur, so that there is exactly one active neuron in each row and column in a valid solution. The weights between adjacent columns encode -1 times the distance between the cities, and the weights between nodes in the same row or column should be set as a large negative value to stop more than one entry in each row and column being activated. Implement this and compare it to solving the TSP using the methods in Section 9.4.

Problem 17.4 Create a dataset that consists of horizontal and vertical stripes in a 2D array of size 4×4. Test whether or not an RBM can differentiate between the horizontal and vertically striped examples. Is the result what you expected?

Problem 17.5 In the RBM the probabilities for the nodes can be used, or actual activations based on random numbers. Hinton, in the practical guide referred to in the Further Reading section suggests that for the hidden nodes it is important to use activations for the hidden nodes (except for the final step of the CD learning), but probabilities are fine for the visible nodes. Modify the code to experiment with using both versions and compare the results on the AlphaDigits dataset.

Problem 17.6 Hinton also suggests that it can be very effective to use minibatches of between 10 and 100 examples (as was discussed in Section 4.2.7 for the MLP) in order to estimate the gradient. Implement this and investigate how many cases work best for different datasets. Make sure that you randomise the order to the data at each iteration.

Problem 17.7 Apply the Deep Belief Network to the MNIST dataset. Compare to just using a single RBM.

CHAPTER 18

Gaussian Processes

The supervised machine learning algorithms that we have seen have generally tried to fit a parametrised function to a set of training data in order to minimise an error function. This function is then used to generalise to previously unseen data. Some of the differences between the methods have been the set of model functions that the algorithm can use to represent the data; for example, the linear models of Chapter 3 and the piecewise constant splines of Chapter 5. However, if we do not know anything about the underlying process that generated the data, then choosing an appropriate model is often a trial-and-error process.

As a very simple example, Figure 18.1 shows a few datapoints. If we assumed that these were drawn from a single Gaussian distribution then we would have two parameters to fit (the mean and standard deviation) in order to get the best match that we could, as shown in the middle figure. However, choosing a different distribution (here, a Weibull distribution, which also has two parameters):

$$f(x;k,\lambda) = \begin{cases} \frac{k}{\lambda}\left(\frac{x}{\lambda}^{k-1} e^{-\left(\frac{x}{\lambda}\right)^{k}}\right), & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{18.1}$$

gives a better fit, as shown on the right (where the dashed line is the Weibull distribution and the solid line is the Gaussian). For the Gaussian $\mu = 0.7$ and $\sigma^2 = 0.25$, while for the Weibull $k = 2$ and $\lambda = 1$.

One possible solution to this problem is to let the optimisation process search over

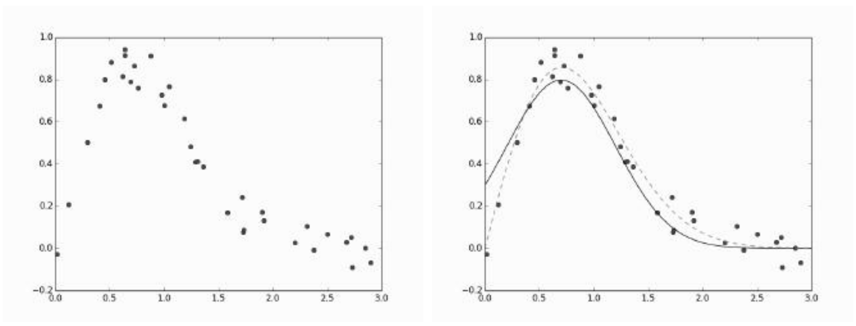

FIGURE 18.1 *Left:* a set of datapoints, *right:* two possible fits to that data, using a Gaussian (solid line) and Weibull distribution (dashed line). It can be seen that the Weibull distribution fits the data better, although both are a fairly good fit.

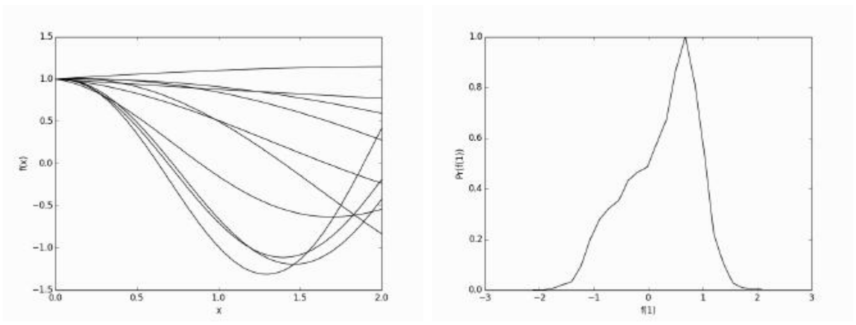

FIGURE 18.2 *Left:* 10 samples from the stochastic process $f(x) = \exp(ax)\cos(bx)$ with a and b drawn from Gaussian distributions. *Right:* The probability distribution of $f(1)$ based on 10,000 samples of $f(x)$.

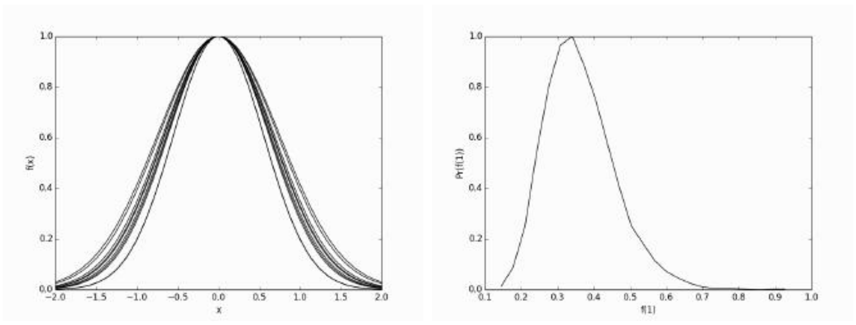

FIGURE 18.3 *Left:* 10 samples from the stochastic process $f(x) = \exp(-ax^2)$ with $a > 0$ drawn from a Gaussian distribution. *Right:* The probability distribution of $f(1)$ based on 10,000 samples of $f(x)$.

different models as well as the parameters of the model. To do this, we need to generalise the idea of a probability distribution to something that we can optimise over. This is known as a stochastic process, and it is simply a collection of random variables put together: instead of having a set of parameters that specify a probability distribution (such as the mean and covariance matrix for a multivariate Gaussian), we have a set of functions and a distribution over that set of functions. Figure 18.2 shows an example of a set of samples from the stochastic process $f(x) = \exp(ax)\cos(bx)$ with a drawn from a Gaussian with mean 0 and variance 0.25, and b from a Gaussian with mean 1 and variance 1, together with the probability distribution of $f(1)$ (computed from a set of 10,000 samples of $f(x)$).

Dealing with general stochastic processes is very difficult because combining the random variables is generally hard. However, if we restrict the process in such a way that all of the random variables have a Gaussian distribution, and the joint distribution over any (finite) subset of the variables is also Gaussian, then this Gaussian process (GP) is much easier to deal with. In order to see that it is still very powerful, Figure 18.3 shows a set of samples from $f(x) = \exp(-ax^2)$ with a drawn from a Gaussian distribution with mean 1 and standard deviation 0.25. It can be seen that the probability distribution of $f(1)$ is not a Gaussian.

The way to think about modelling with a Gaussian process is that we put a probability distribution over the space of functions and sample from that. A function is a mapping from some (possibly multi-dimensional) input $\mathbf{x}$ to $f(\mathbf{x})$, so to specify the function we could just list the value of $f(x)$ for every value of x, which would be an infinitely long vector. One sample would consist of a specification of this vector. However, because everything is Gaussian, just as we specify a Gaussian distribution with the mean and covariance matrix, we can specify a Gaussian process by the mean function and a covariance function.

A complete specification of a particular function would, as has already been remarked, require an infinitely long vector. However, it turns out that Gaussian processes are very well behaved, so that considering only finite sets of points gives exactly the same inference result as would the compete integral (for more on this, see the references in the Further Reading section).

There have been various versions of Gaussian processes around for a very long time, known as kriging after one inventor, and Kolmogorov–Wiener prediction after two more. In more than one dimension it is technically a Gaussian random field, which was the focus of Section 16.2.

In fact, Gaussian processes are just smoothers, fitting a smooth curve through a set of datapoints. It seems amazing that such a simple process can be so powerful, but regression problems do all pretty much boil down to finding a smooth function that passes through the data. More surprisingly, we will see later in the chapter that Gaussian processes can also solve classification problems, which are harder to view in this way. Regardless of how it is viewed, it is time to start working out how to use one.

18.1 GAUSSIAN PROCESS REGRESSION

As was mentioned previously, the GP is specified by the mean and covariance functions. In fact, it is usual to subtract off the mean first, so that the mean function is identically zero. In this case, the GP is completely described as a function $G(k(\mathbf{x}, \mathbf{x}'))$ that models some underlying function $f(\mathbf{x})$, where covariance function $k(\mathbf{x}, \mathbf{x}')$ gives us the expected covariance matrix between the values of f at $\mathbf{x}$ and $\mathbf{x}'$. The random variables that define the GP are used to provide an estimate of $f(\mathbf{x})$ for each input $\mathbf{x}$.

This is where we have to put some prior work in: the covariance function needs to be specified, and this is what provides the expressive power of the GP. This covariance function is the same thing as the kernel, which we explored in Chapter 8, and there are strong links between SVMs and GPs; for more details see the references in the Further Reading section.

Taking a hint from SVMs, then, we will start with a covariance matrix that has the form of the RBF kernel:

$$k(\mathbf{x}, \mathbf{x}') = \sigma_f^2 \exp\left(-\frac{1}{2l^2}|\mathbf{x} - \mathbf{x}'|^2\right). \tag{18.2}$$

In GPs, for some reason, this is normally known as the squared exponential covariance matrix rather than the RBF. For a set of input vectors it enables us to specify a matrix of covariances $\mathbf{K}$ where the element at place (i, j) in the matrix is $K_{ij} = k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$.

There are two parameters in this covariance function: σ_f and l, and we shall consider them shortly. First, though, we will work out how to use the GP to predict the value of $f^* = f(\mathbf{x}^*)$ for some values of $\mathbf{x}^*$ based on a training set of values $f(\mathbf{x})$.

As is usual for supervised learning, the training set consists of a set of N labelled examples $(\mathbf{x}_i, t_i), i = 1..N$. Since this is a GP, the joint density $P(t^*, \mathbf{t}_N)$ is a Gaussian (where the notation is meant to imply that t^* is a single test point, while $\mathbf{t}_N$ is the whole set of training target labels) and so is this conditional distribution:

$$P(t^*|\mathbf{t}_N) = P(t^*, \mathbf{t}_N)/P(\mathbf{t}_N). \tag{18.3}$$

The covariance matrix for the joint distribution is $\mathbf{K}_{N+1}$, which has size $(N+1)\times(N+1)$, and can be partitioned in the following way:

$$\mathbf{K}_{N+1} = \begin{pmatrix} \begin{bmatrix} & \\ & \mathbf{K}_N & \\ & \end{bmatrix} & \begin{bmatrix} \\ \mathbf{k}^* \\ \\ \end{bmatrix} \\ \begin{bmatrix} & \mathbf{k}^{*T} & \end{bmatrix} & \begin{bmatrix} k^{**} \end{bmatrix} \end{pmatrix} \tag{18.4}$$

where $\mathbf{K}_N$ is the covariance matrix for the training data, $\mathbf{k}^*$ is the covariance matrix between the test points $\mathbf{x}^*$ and the training data (which also appears in transposed form), and k^{**} is the covariance between the points in the test set (which will be a single scalar value when building $\mathbf{K}_{N+1}$ from $\mathbf{K}_N$). If there are N pieces of training data and n test points, then the sizes of these parts are $N \times N$, $N \times n$, and $n \times n$, respectively. We will drop the size subscript from $\mathbf{K}$ from now on, and use it to denote the covariance matrix of the training data ($\mathbf{K}_N$) and use the notation introduced in Equation (18.4).

The joint distribution of the training and test data ($p(\mathbf{t}, t^*)$ is the Gaussian distribution with zero mean and the extended covariance matrix shown in Equation (18.4). We know the values of the observations for the test data, so we only want to produce samples that match the observables at these points. We could do this by choosing random samples and throwing them away if they don't match, but this would be very slow, since very few of the samples would match.

Fortunately, we can condition the joint distribution on the training data, which gives us the posterior distribution as:

$$P(t^*|\mathbf{t}, \mathbf{x}, \mathbf{x}^*) \propto \mathcal{N}\left(\mathbf{k}^{*T}\mathbf{K}^{-1}\mathbf{t}, k^{**} - \mathbf{k}^{*T}\mathbf{K}^{-1}\mathbf{k}^*\right), \tag{18.5}$$

where $\mathcal{N}(m, \Sigma)$ denotes a Gaussian distribution with mean m and covariance Σ.

There is one important thing to notice, which is the requirement to invert the $N \times N$ matrix $\mathbf{K}$, which is an expensive operation, and not necessarily a numerically stable one. The good news is that only the covariance matrix of the training data needs to be inverted, and so this only has to be done once. However, if there is a lot of training data then this is still an expensive $\mathcal{O}(N^3)$ operation, and it requires that the matrix is (numerically) invertible.

18.1.1 Adding Noise

The top-left plot in Figure 18.4 shows the mean and plus/minus two standard deviations of the posterior distribution for the squared exponential kernel with the five datapoints marked as the training data. In that plot, you can see that the variance at the training data is zero, which is fine if you don't believe that your training data has any noise. However, this is, of course, very unlikely. The usual way to add noise into any GP is to assume that it is independent, identically distributed Gaussian noise and so include an extra parameter into the covariance matrix, so that instead of using $\mathbf{K}$ we use $\mathbf{K}+\sigma_n^2\mathbf{I}$, where $\mathbf{I}$ is the $N \times N$ identity matrix. Noise is only added to the covariance for the training data. Together, the parameters of the kernel, including σ_n, are known as hyperparameters.

The posterior distribution is then:

$$P(t^*|\mathbf{t}, \mathbf{x}, \mathbf{x}^*) \propto \mathcal{N}\left(\mathbf{k}^{*T}(\mathbf{K} + \sigma_n\mathbf{I})^{-1}\mathbf{t}, k^{**} - \mathbf{k}^{*T}(\mathbf{K} + \sigma_n\mathbf{I})^{-1}\mathbf{k}^*\right). \tag{18.6}$$

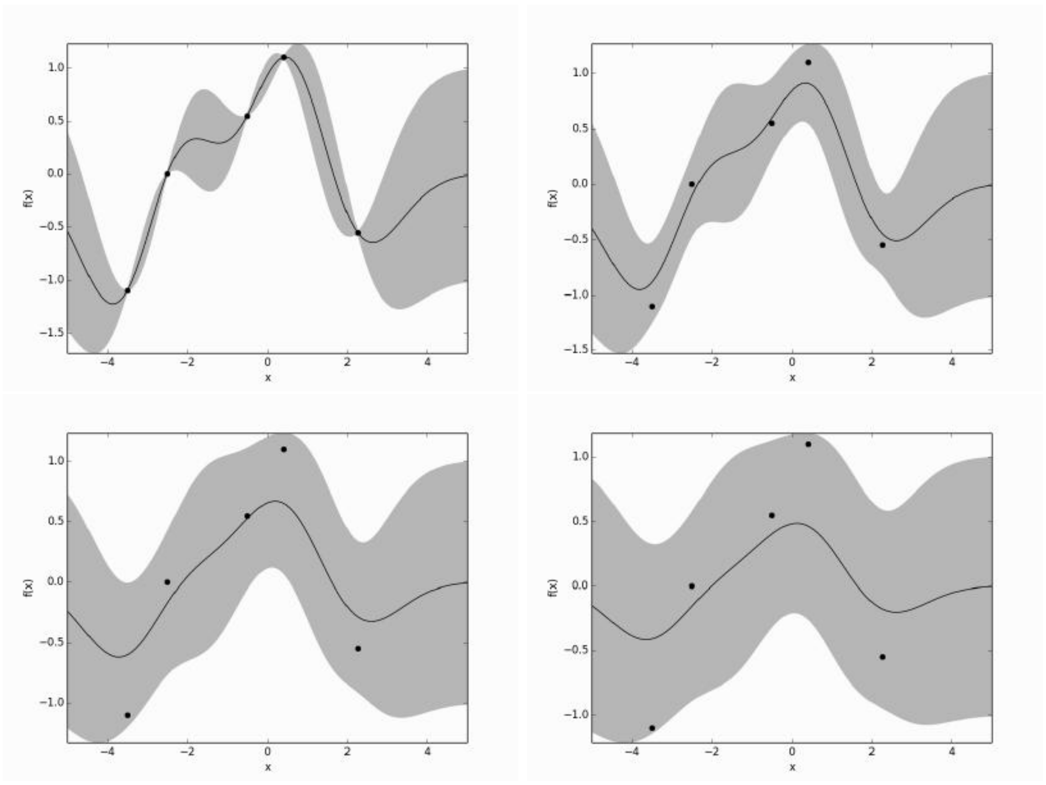

FIGURE 18.4 The effects of adding noise to the estimate of the covariance in the training data with the squared exponential kernel. Each plot shows the mean and 2 standard deviation error bars for a Gaussian process fitted to the five datapoints marked with dots. *Top left:* $\sigma_n = 0.0$, *top right:* $\sigma_n = 0.2$, *bottom left:* $\sigma_n = 0.4$, *bottom right:* $\sigma_n = 0.6$

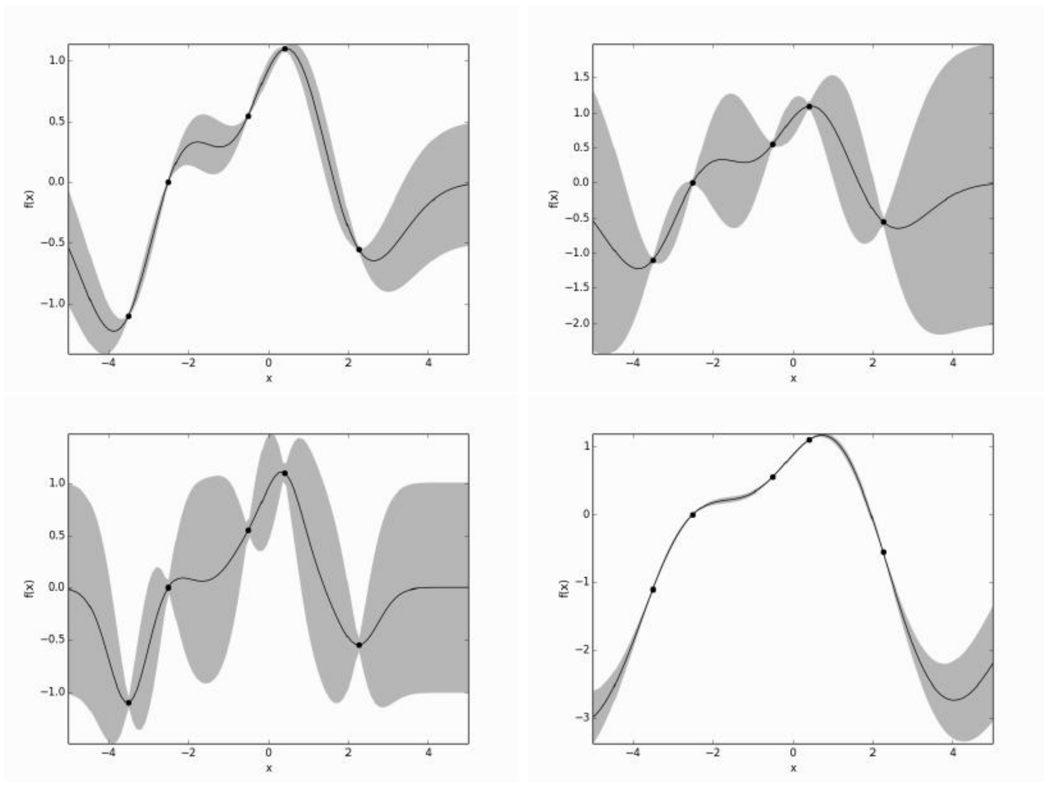

FIGURE 18.5 The effects of the other two parameters in the squared exponential kernel (compare to the top-left plot of Figure 18.4). Each plot shows the mean and 2 standard deviation error bars for a Gaussian process fitted to the five datapoints marked with dots. The parameters of the kernels were: *Top left:* $\sigma_f = 0.25, l = 1.0, \sigma_n = 0.0$, *top right:* $\sigma_f = 1.0, l = 1.0, \sigma_n = 0.0$, *bottom left:* $\sigma_f = 0.5, l = 0.5, \sigma_n = 0.0$, *bottom right:* $\sigma_f = 0.5, l = 2.0, \sigma_n = 0.0$.

The other three plots in Figure 18.4 show the effect that adding increasing amounts of observation noise makes.

Since we have considered the role of one of the hyperparameters, this is also a good place to consider the role of σ_f and l. Figure 18.5 shows the effects of changing these parameters for the same data as in Figure 18.4. It can be seen that modifying the signal variance σ_f^2 simply controls the overall variance of the function, while the length scale l changes the degree of smoothing, trading it off against how well the curve matches the training data.

Of the two parameters it is the l factor that is of most interest. It acts as a length scale, which says something about how quickly the function changes as the inputs vary. Figure 18.6 shows GP regression with similar data, except that in the plots on the second row, the x values of the points have been brought closer together. On the left, $l = 1.0$, while on the right $l = 0.5$. It can be seen that the top left and bottom right plots, where the length scale 'matches' the distances in the data, the fit looks smoother.

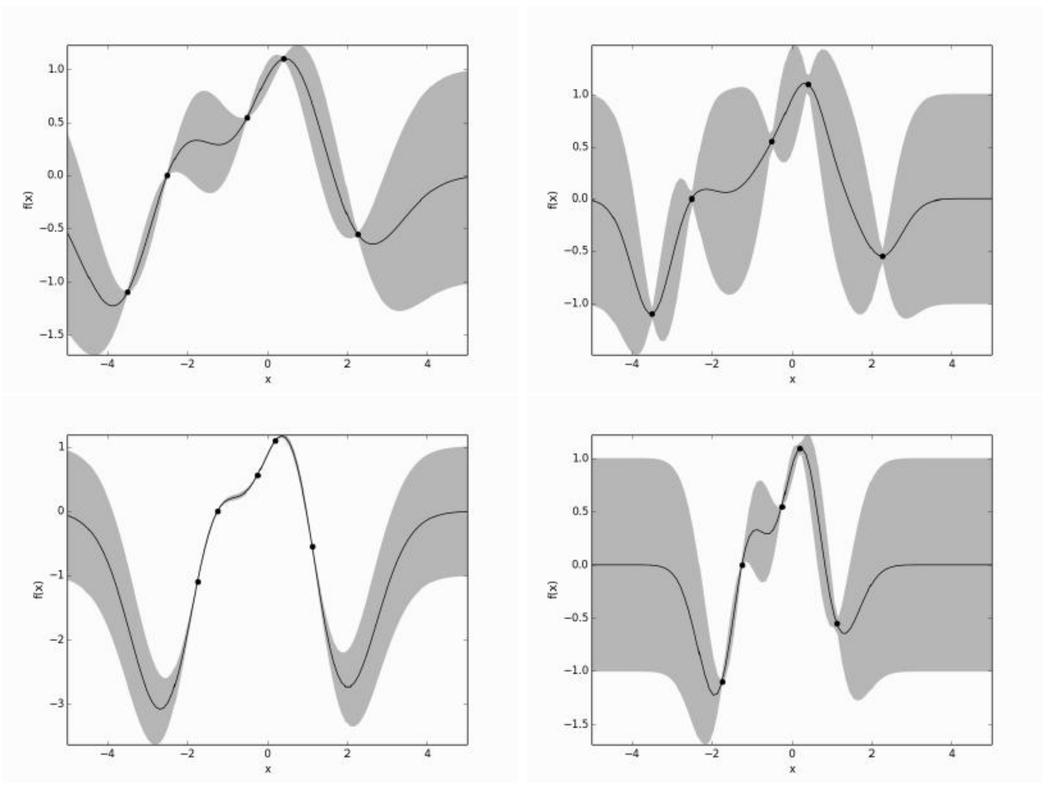

FIGURE 18.6 The effects of changing the length scale in GP regression. The top row shows one dataset, while the second row shows the same dataset, but with the points brought closer together. The length scale is the same for the plots above each other, being $l = 1.0$ on the left and $l = 0.5$ on the right.

18.1.2 Implementation

We have seen everything that we need to compute a basic Gaussian process regression program: we compute the covariance matrix of the training data, and also the covariances between the training and test data, and the test data alone. Then we compute the mean and covariance of the posterior distribution and sample from it. This results in the following algorithm:

Gaussian Process Regression

- For given training data $(\mathbf{X}, \mathbf{t})$, test data $\mathbf{x}^*$, covariance function $k()$, and hyperparameters $\boldsymbol{\theta} = (\sigma_f^2, l\sigma_n^2)$:
  - compute the covariance matrix $\mathbf{K} = k(\mathbf{X}, \mathbf{X}) + \sigma_n \mathbf{I}$ for hyperparameters $\boldsymbol{\theta}$
  - compute the covariance matrix $\mathbf{k}^* = k(\mathbf{X}, \mathbf{x}^*)$
  - compute the covariance matrix $k^{**} = k(\mathbf{x}^*, \mathbf{x}^*)$
  - the mean of the process is $\mathbf{k}^{*T}\mathbf{K}^{-1}\mathbf{t}$
  - the covariance is $k^{**} - \mathbf{k}^{*T}\mathbf{K}^{-1}\mathbf{k}^*$

However, before implementing it, there are a few numerical problems that need to be dealt with, as inverting the matrix $(\mathbf{K}+\sigma_n\mathbf{I})$ is not always stable, as it can have eigenvalues that are very close to 0.

Since we know that $\mathbf{K}$ is symmetric and positive definite, there are more stable ways to perform the inversion. The key is what is known as the Cholesky decomposition, which decomposes a real-valued matrix $\mathbf{K}$ into the product $\mathbf{L}\mathbf{L}^T$, where $\mathbf{L}$ is a lower triangular matrix that only has non-zeros entries on and below the leading diagonal. There are two benefits to this, first that it is relatively cheap to calculate the inverse of a lower triangular matrix (and the inverse of the original matrix is $\mathbf{K}^{-1} = \mathbf{L}^{-T}\mathbf{L}^{-1}$, where $\mathbf{L}^{-T} = (\mathbf{L}^{-1})^T$), and secondly that it provides a very quick and easy way to solve linear systems $\mathbf{A}\mathbf{x} = \mathbf{b}$.

In fact, these two benefits are both parts of the same thing, since the inverse of a matrix $\mathbf{A}$ is the matrix $\mathbf{B}$ for which $\mathbf{AB} = \mathbf{I}$, and we can solve this column-by-column as $\mathbf{AB}_i = \mathbf{I}_i$ (where the subscript is an index for the ith column of the matrix).

To solve $\mathbf{L}\mathbf{L}^T\mathbf{x} = \mathbf{t}$ it is simply a matter of forward substitution to find the $\mathbf{z}$ that solves $\mathbf{Lz} = \mathbf{t}$ followed by back-substitution to find the $\mathbf{x}$ that solves $\mathbf{L}^T\mathbf{x} = \mathbf{z}$.

The cost of these operations is $\mathcal{O}(n^3)$ for the Cholesky decomposition and $\mathcal{O}(n^2)$ for the solve, and the whole thing is numerically very stable. NumPy provides implementations of both of these computations in the `np.linalg` module, and so the whole computation of the mean (`f`) and covariance (`V`) can be written as:

```
L = np.linalg.cholesky(k)
beta = np.linalg.solve(L.transpose(), np.linalg.solve(L,t))
kstar = kernel(data,xstar,theta,wantderiv=False,measnoise=0)
f = np.dot(kstar.transpose(), beta)
v = np.linalg.solve(L,kstar)
V =                          kernel(xstar,xstar,theta,wantderiv=False,measnoise=0)-
np.dot(v.transpose(↩
),v)
```

The computation of `V` uses $\mathbf{v}^T\mathbf{v}$, where $\mathbf{L}\mathbf{v} = \mathbf{k}^*$ and to see that this does indeed match the covariance in Equation (18.6) requires a little bit of algebra:

$$\begin{aligned}\mathbf{k}^{*T}\mathbf{K}^{-1}\mathbf{k}^* &= (\mathbf{L}\mathbf{v})^T\mathbf{K}^{-1}\mathbf{L}\mathbf{v} \\ &= \mathbf{v}^T\mathbf{L}^T(\mathbf{L}\mathbf{L}^T)^{-1}\mathbf{L}\mathbf{v} \\ &= \mathbf{v}^T\mathbf{L}^T\mathbf{L}^{-T}\mathbf{L}^{-1}\mathbf{L}\mathbf{v} \\ &= \mathbf{v}^T\mathbf{v}\end{aligned}$$

Comparing the code to Equation (18.6) you might also notice that the mean can be written in a slightly different way as:

$$m(\mathbf{x}, \mathbf{x}^*) = \sum_i \beta_i k(\mathbf{x}_i, \mathbf{x}^*), \tag{18.7}$$

where β_i is the ith part of $\beta = (\mathbf{K} + \sigma_n^2\mathbf{I})^{-1}\mathbf{t}$. This suggests that we can consider GP regression as the sum of a set of basis functions positioned on the training data; indeed for the squared exponential covariance matrix, we have produced precisely an RBF method; see Chapter 5. In that chapter we could modify the weights that specified the locations of the RBFs, but here we can't, but we can modify the weights that connect them to the outputs. Seen in this way, this GP is basically a linear neural network.

So providing that the hyperparameters are chosen to match the data, using a GP for regression is very simple. Now we are ready to do some learning to modify the parameters based on the data in order to improve the fit of the GP.

18.1.3 Learning the Parameters

The squared exponential covariance matrix (Equation (18.2)) has three hyperparameters (σ_f, σ_n, l) that need to be selected, and we have already seen that they can have a significant effect on the shape of the resulting output curve, so that finding the correct values is very important. In the next section we will also see that with more complex covariance matrices there are many more hyperparameters to choose, and so finding an automatic method of choosing the hyperparameters is clearly important if GPs are going to be useful.

If the set of hyperparameters are labelled as $\boldsymbol{\theta}$ then the ideal solution to this problem would be to set up some kind of prior distribution over the hyperparameters and then integrate them out in order to maximise the probability of the output targets:

$$P(t^*|\mathbf{x}, \mathbf{t}, \mathbf{x}^*) = \int P(t^*|\mathbf{x}, \mathbf{t}, \mathbf{x}^*, \boldsymbol{\theta})P(\boldsymbol{\theta}|\mathbf{x}, \mathbf{t})d\boldsymbol{\theta}. \tag{18.8}$$

This integral is very rarely tractable, but we can compute the posterior probability of $\boldsymbol{\theta}$ (which is the marginal likelihood times $P(\boldsymbol{\theta})$). The log of the marginal likelihood (also known as the evidence for the hyperparameters, which marginalises over the function values) is:

$$\log P(\mathbf{t}|\mathbf{x}, \boldsymbol{\theta}) = -\frac{1}{2}\mathbf{t}^T(\mathbf{K} + \sigma_n^2\mathbf{I})^{-1}\mathbf{t} - \frac{1}{2}\log|\mathbf{K} + \sigma_n^2\mathbf{I}| - \frac{N}{2}\log 2\pi. \tag{18.9}$$

In order to derive this equation you need to remember that the product of two Gaussians is also Gaussian (up to normalisation) and then write out the equation of a multivariate Gaussian and take the logarithm.

We now want to minimise this log likelihood, which we can do by using our favourite gradient descent solver from Chapter 9 (for example, conjugate gradients from Section 9.3), providing that we first compute the gradient of it with respect to each of the hyperparameters. We will write $\mathbf{Q} = (\mathbf{K} + \sigma_n^2\mathbf{I})$ and then recall that $\mathbf{Q}$ is a function of all of the hyperparameters $\boldsymbol{\theta}_i$. Amazingly, these derivatives have a very nice form, as can be seen with the use of two matrix identities (where $\frac{\partial Q}{\partial \boldsymbol{\theta}}$ is simply the element-by-element derivative of the matrix):

$$\begin{aligned}\frac{\partial \mathbf{Q}^{-1}}{\partial \boldsymbol{\theta}} &= -\mathbf{Q}^{-1}\frac{\partial \mathbf{Q}}{\partial \boldsymbol{\theta}}\mathbf{Q}^{-1} &(18.10)\\ \frac{\partial \log|\mathbf{Q}|}{\partial \boldsymbol{\theta}} &= \text{trace}\left(\mathbf{Q}^{-1}\frac{\partial \mathbf{Q}}{\partial \boldsymbol{\theta}}\right). &(18.11)\end{aligned}$$

Then:

$$\frac{\partial}{\partial \boldsymbol{\theta}}\log P(\mathbf{t}|\mathbf{x},\boldsymbol{\theta}) = \frac{1}{2}\mathbf{t}^T\mathbf{Q}^{-1}\frac{\partial Q}{\partial \boldsymbol{\theta}}\mathbf{Q}^{-1}\mathbf{t} - \frac{1}{2}\text{trace}\left(\mathbf{Q}^{-1}\frac{\partial \mathbf{Q}}{\partial \boldsymbol{\theta}}\right). \tag{18.12}$$

Now, all that is required is to actually perform the computations of the derivatives of the covariance with respect to each hyperparameter, and then optimise the log likelihood using the conjugate gradient solver.

It will make things slightly easier if we change the way that the hyperparameters are presented a little bit. Note that all of the hyperparameters are positive numbers (since they are all squared in Equation (18.2). We can also make them positive by taking the exponential of each of them, and since the derivative of an exponential is just the exponential, this can make things a little clearer. Further, we will effectively work with $1/\sigma_l$ since it also makes the computation easier.

For the squared exponential kernel (where there is a slight notation abuse in the use of the identity matrix $\mathbf{I}$ in the last term):

$$\begin{aligned}k(\mathbf{x},\mathbf{x}') &= \exp(\sigma_f)\exp\left(-\frac{1}{2}\exp(\sigma_l)|\mathbf{x}-\mathbf{x}'|^2\right) + \exp(\sigma_n)\mathbf{I} &(18.13)\\ &= k' + \exp(\sigma_n)\mathbf{I} &(18.14)\end{aligned}$$

these are nice and easy to compute:

$$\frac{\partial k}{\partial \sigma_f} = k' \tag{18.15}$$

$$\frac{\partial k}{\partial \sigma l} = k' \times \left(-\frac{1}{2}\exp(\sigma_l)|\mathbf{x}-\mathbf{x}'|^2\right) \tag{18.16}$$

$$\frac{\partial k}{\partial \sigma_n} = \exp(\sigma_n)\mathbf{I} \tag{18.17}$$

Note that the term inside the bracket in $\frac{\partial k}{\partial \sigma l}$ is precisely the one that has already been computed for the exponential calculation.

18.1.4 Implementation

The basic algorithm is very simple again, which is to call the conjugate gradient optimiser to minimise the log likelihood, providing it with the computations of the gradients with respect to the parameters. The SciPy optimiser was used in Section 9.3 and the syntax is no different here:

```
result = so.fmin_cg(logPosterior, theta, fprime=gradLogPosterior,
args=[(X,y)⤸
], gtol=1e-4,maxiter=5,disp=1)
```

where possible implementations of the log likelihood and gradient functions are:

```
def logPosterior(theta,args):
        data,t = args
        k = kernel2(data,data,theta,wantderiv=False)
        L = np.linalg.cholesky(k)
        beta = np.linalg.solve(L.transpose(), np.linalg.solve(L,t))
        logp = -0.5*np.dot(t.transpose(),beta) - np.sum(np.log(np.⤸
        diag(L))) - np.shape(data)[0] /2. * np.log(2*np.pi)
        return -logp

def gradLogPosterior(theta,args):
        data,t = args
        theta = np.squeeze(theta)
        d = len(theta)
        K = kernel2(data,data,theta,wantderiv=True)

        L = np.linalg.cholesky(np.squeeze(K[:,:,0]))
        invk = np.linalg.solve(L.transpose(),np.linalg.solve(L,np.⤸
        eye(np.shape(data)[0])))

        dlogpdtheta = np.zeros(d)
        for d in range(1,len(theta)+1):
                dlogpdtheta[d-1] = 0.5*np.dot(t.transpose(), np.dot(⤸
                invk, np.dot(np.squeeze(K[:,:,d]), np.dot(invk,t)))) ⤸
                - 0.5*np.trace(np.dot(invk,np.squee
ze(K[:,:,d])))

        return -dlogpdtheta
```

In terms of implementation, the only thing that we have not covered yet is how to compute the covariance matrix, but there is nothing complex about that: the function takes in two sets of datapoints and returns the covariance matrix, and possibly the gradients as well (which is the `wantd` switch). One possible way to do this for the squared exponential kernel is:

```
def kernel(data1,data2,theta,wantderiv=True,measnoise=1.):
        # Squared exponential
        theta = np.squeeze(theta)
        theta = np.exp(theta)
        if np.ndim(data1) == 1:
                d1 = np.shape(data1)[0]
                n = 1
        else:
                (d1,n) = np.shape(data1)

        d2 = np.shape(data2)[0]
        sumxy = np.zeros((d1,d2))
        for d in range(n):
                D1 = np.transpose([data1[:,d]]) * np.ones((d1,d2))
                D2 = [data2[:,d]] * np.ones((d1,d2))
                sumxy += (D1-D2)**2*theta[d+1]

        k = theta[0] * np.exp(-0.5*sumxy)

        if wantderiv:
                K = np.zeros((d1,d2,len(theta)+1))
                K[:,:,0] = k + measnoise*theta[2]*np.eye(d1,d2)
                K[:,:,1] = k
                K[:,:,2] = -0.5*k*sumxy
                K[:,:,3] = theta[2]*np.eye(d1,d2)
                return K
        else:
                return k + measnoise*theta[2]*np.eye(d1,d2)
```

Figure 18.7 shows an example of a Gaussian process before and after optimisation, with initially random hyperparameters. Before the optimisation process the log likelihood of the data under the model, based on random initialisation of the hyperparameters, was around 60, whereas afterwards it was around 16. It can be seen that the model fits the data much better after the optimisation process.

18.1.5 Choosing a (set of) Covariance Functions

Like any other kernel, the choice of covariance function is crucial to successful prediction. This is the modelling part of GP learning, and it is entirely human-dependent: you choose appropriate covariance functions and then the algorithm learns their parameters. All that is required is that the functions must generate positive-definite (or actually, non-negative definite) covariance matrices. There is a fairly large choice of typical kernels for GPs, but given that our only restriction is that they must be positive-definite, it is possible to add and multiply kernels as we saw in Section 8.2 using Mercer's theorem. The upshot of this is that you can string together a whole set of covariance functions that represent different parts of what you believe the data is doing. So for example, if you think that there are two different squared exponential processes, but with different length scales, you could include two versions of the kernel, and optimise the two different length scales.

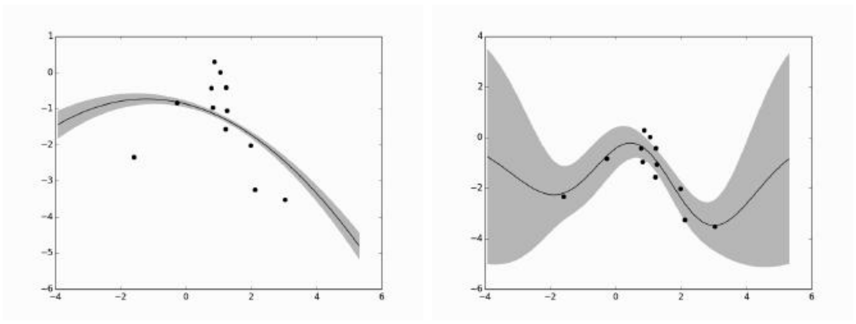

FIGURE 18.7 *Left:* the data and the model based on random parameters, *right:* the fitted model.

A few commonly used covariance functions are:

Constant $k(\mathbf{x}, \mathbf{x}') = e^{\sigma}$

Linear $k(\mathbf{x}, \mathbf{x}') = \sum_{d=1}^{D} e^{\sigma_d} \mathbf{x}_d \mathbf{x}'_d$

Squared Exponential $k(\mathbf{x}, \mathbf{x}') = e^{\sigma_f} \exp\left(-\frac{1}{2}\exp(\sigma_l)(\mathbf{x} - \mathbf{x}')^2\right)$

Ornstein–Uhlenbeck $k(\mathbf{x}, \mathbf{x}') = \exp\left(-\exp(\sigma_l)|\mathbf{x} - \mathbf{x}'|\right)$

Matérn $k(\mathbf{x}, \mathbf{x}') = \frac{1}{2^{\sigma_\nu - 1}\Gamma(\sigma_\nu)}\left(\frac{\sqrt{2\sigma_\nu}}{l}(\mathbf{x} - \mathbf{x}')\right)^{\nu} K_\nu\left(\frac{\sqrt{2\sigma_\nu}}{l}(\mathbf{x} - \mathbf{x}')\right)$,
where K_{σ_ν} is a modified Bessel function and Γ is the gamma function.

Periodic $k(\mathbf{x}, \mathbf{x}') = \exp\left(-2\exp(\sigma_l)\sin^2(\sigma_\nu \pi(\mathbf{x} - \mathbf{x}'))\right)$

Rational Quadratic $k(\mathbf{x}, \mathbf{x}') = \left(1 + \frac{1}{2\sigma_\alpha}\exp(\sigma_l)(\mathbf{x} - \mathbf{x}')^2\right)^{-\sigma_\alpha}$

18.2 GAUSSIAN PROCESS CLASSIFICATION

While it is possible to perform multi-class classification with a Gaussian process, we will consider only two classes, labelled as +1 and -1. The task of the process is then to model the probability that input $\mathbf{x}$ belongs to class 1, which means that the output should be a value between 0 and 1 (inclusive) like all good probabilities. We will arrange this in the same way that we did it for neurons: by squashing it using the logistic function $P(t^* = 1|a) = \sigma(a) = 1/(1 + \exp(-a))$, where a is the output of the regression GP, and a little care is needed since we are now using $\sigma(\cdot)$ to denote the logistic function, as well as σ_n to denote a hyperparameter and even σ^2 as the variance. Since there are two classes $P(t^* = -1|a) = 1 - P(t^* = 1|a)$, and so we can write $p(t^*|a) = \sigma(t^* f(x^*))$. So GP classification consists of finding a GP prior over $f(\mathbf{x})$ (known as the latent function) and then putting this through the logistic function to find a prior on the predicted class, which is:

$$p(t^* = 1|\mathbf{x}, \mathbf{t}, \mathbf{x}^*) = \int \sigma(f(\mathbf{x}^*)) p(f(\mathbf{x}^*)|\mathbf{x}, \mathbf{t}, \mathbf{x}^*) df(\mathbf{x}^*). \tag{18.18}$$

This is a 1D integral, and so it can be computed numerically, but unfortunately the likelihood function $p(f(\mathbf{x}^*)|\mathbf{x}, \mathbf{t}, \mathbf{x}^*)$ is not a Gaussian function and so computing that term is intractable. This means that some form of approximation is needed. There are several methods of doing these approximations, including using MCMC, but we will consider only the simplest version, which is known as Laplace's approximation. The references at the end of the chapter provide a list of places with more information about more advanced approximation methods.

18.2.1 The Laplace Approximation

Laplace's approximation is a way to approximate any integral of the form $\int \exp(f(\mathbf{x}))d\mathbf{x}$, which, of course, includes Gaussians. The basic idea is to find the global maximum of the function $f(\mathbf{x})$, which occurs at some $\mathbf{x}_0$. At this point the gradient of $f(\mathbf{x})$ (which is $\nabla f(\mathbf{x})$) is 0, and so the second-order Taylor expansion around $\mathbf{x}_0$ is (where $\nabla\nabla f(\cdot)$ is the Hessian matrix):

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \frac{1}{2}(\mathbf{x}-\mathbf{x}_0)^T\nabla\nabla f(\mathbf{x})(\mathbf{x}-\mathbf{x}_0). \tag{18.19}$$

Since the logarithm of a Gaussian is a quadratic function, this has a unique maximum, and so we replace $f(\mathbf{x})$ by $\log f(\mathbf{x})$ and then compute the exponential of this, which tells us that:

$$f(\mathbf{x}) \approx f(\mathbf{x}_0)\exp\left(\frac{1}{2}(\mathbf{x}-\mathbf{x}_0)^T\nabla\nabla\log f(\mathbf{x})(\mathbf{x}-\mathbf{x}_0)\right). \tag{18.20}$$

Normalising this to make it a Gaussian distribution tells us that:

$$\begin{aligned} q(f(\mathbf{x})|\mathbf{x},\mathbf{t}) &\propto \exp\left(-\frac{1}{2}(f(\mathbf{x})-\hat{f}(\mathbf{x}))^T\mathbf{W}(f(\mathbf{x})-\hat{f}(\mathbf{x}))\right) \\ &= \mathcal{N}(f(\mathbf{x})|f(\mathbf{x}_0,\mathbf{W}^{-1})), \end{aligned} \tag{18.21}$$

where $\mathbf{W} = -\nabla\nabla\log f(\mathbf{x})$.

In order to compute the Laplace approximation we need to find the value of $\mathbf{x}_0$ and then evaluate the Hessian matrix at that point. Identifying $\mathbf{x}_0$ can be done using the Newton–Raphson iteration, which finds an approximation to solutions of $f(x) = 0$ (in fact, here we want to find $f'(x) = 0$, but this doesn't change things much) by iterating the computation:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{18.22}$$

until the changes are sufficiently small for the required accuracy.

18.2.2 Computing the Posterior

Returning to the actual computations that we need for the GP, we had reached the stage of approximating $p(f(\mathbf{x}^*)|\mathbf{x},\mathbf{t},\mathbf{x}^*)$. Using Bayes' rule we get that:

$$p(f(\mathbf{x})|\mathbf{x},\mathbf{t}) = \frac{p(\mathbf{t}|f(\mathbf{x}))p(f(\mathbf{x})|\mathbf{x})}{p(\mathbf{t}|\mathbf{x})}. \tag{18.23}$$

We are in the lucky situation that the denominator is independent of $f()$, and so can be ignored for the optimisation. The first term in the numerator is:

$$p(\mathbf{t}|f(\mathbf{x})) = \prod_{i=1}^{N} \sigma(f(\mathbf{x}_i))^{t_n}(1-\sigma(f(\mathbf{x}_i))^{1-t_n}. \tag{18.24}$$

We will need to differentiate the log of this expression twice in order to use Equation (18.21):

$$\begin{aligned} \nabla \log p(\mathbf{t}|f(\mathbf{x})) &= \mathbf{t} - \sigma(f(\mathbf{x})) - \mathbf{K}^{-1}f(\mathbf{x}) & (18.25)\\ \nabla\nabla \log p(\mathbf{t}|f(\mathbf{x})) &= -\text{diag}(\sigma(f(\mathbf{x}))(1-\sigma(f(\mathbf{x})))) - \mathbf{K}^{-1}, & (18.26) \end{aligned}$$

where diag() puts the values along the diagonal of a zero matrix, and this term is the **W** matrix in Equation (18.21).

We now need to find the maximum of $\log p(\mathbf{t}|f(\mathbf{x}))$, for which we construct the Newton–Raphson iteration:

$$\begin{aligned} f(\mathbf{x})^{\text{new}} &= f(\mathbf{x}) - \nabla\nabla \log p(\mathbf{t}|f(\mathbf{x}))\\ &= f(\mathbf{x}) + (\mathbf{K}^{-1}+\mathbf{W})^{-1}(\nabla \log p(\mathbf{t}|f(\mathbf{x})) - \mathbf{K}^{-1}f(\mathbf{x}))\\ &= (\mathbf{K}^{-1}+\mathbf{W})^{-1}(\mathbf{W}f((x)) + \nabla \log p(\mathbf{t}|f(\mathbf{x}))). & (18.27) \end{aligned}$$

Thus, the Laplace approximation to the posterior probability is:

$$q(f(\mathbf{x})|\mathbf{x},\mathbf{t}) = \mathcal{N}(\hat{f}, (\mathbf{K}^{-1}+\mathbf{W})^{-1}). \tag{18.28}$$

Based on this, we can estimate the posterior mean and variance. For the mean, we need to use the fact that at the maximum of $\log p(\mathbf{t}|f(\mathbf{x}))$:

$$\hat{f}(\mathbf{x}) = \mathbf{K}(\nabla \log p(\mathbf{t}|\hat{f}(\mathbf{x}))), \tag{18.29}$$

and then the expressions for the mean and variance of the GP regression give us posterior distribution:

$$P(t^*|\mathbf{t},\mathbf{x},\mathbf{x}^*) \propto \mathcal{N}\left(\mathbf{k}^{*T}(\mathbf{t}-\sigma(f(\mathbf{x}))), k^{**} - \mathbf{k}^{*T}(\mathbf{K}+\mathbf{W}^{-1})^{-1}\mathbf{k}^*\right) \tag{18.30}$$

For the optimisation we will also need to calculate the log likelihood and the gradient of it with respect to each hyperparameter, just as we did for GP regression.

The log likelihood is:

$$\log p(\mathbf{t}|\mathbf{x},\boldsymbol{\theta}) = \int p(\mathbf{t}|f(\mathbf{x})p(f(\mathbf{x})|\boldsymbol{\theta})df(\mathbf{x}), \tag{18.31}$$

and so we again use the Laplace approximation to get:

$$\begin{aligned} \log p(\mathbf{t}|\mathbf{x},\boldsymbol{\theta}) &\approx \log q(\mathbf{t}|\mathbf{x},\boldsymbol{\theta})\\ &= \log p(\hat{f}(\mathbf{x})|\boldsymbol{\theta}) + \log p(\mathbf{t}|\hat{f}(\mathbf{x}) - \frac{1}{2}\log|\mathbf{W}+\mathbf{K}^{-1}| + \frac{N}{2}\log(2\pi). \end{aligned} \tag{18.32}$$

Differentiating this with respect to each of the hyperparameters will lead to two terms, since both $\hat{f}()$ and $\mathbf{K}$ depend on $\boldsymbol{\theta}$. The same matrix identities as for the regression case are useful, and the first part, which is the explicit dependence upon any element of $\boldsymbol{\theta}$ is fairly similar to the regression case:

$$\frac{\partial}{\partial \boldsymbol{\theta}_j} \log p(\mathbf{t}|\boldsymbol{\theta})\bigg|_{\text{explicit}} = \frac{1}{2}\hat{f}(\mathbf{x})^T \mathbf{K}^{-1} \frac{\partial \mathbf{K}}{\partial \boldsymbol{\theta}_j} \mathbf{K}^{-1} \hat{f}(\mathbf{x}) - \frac{1}{2}\text{trace}\left((\mathbf{I} + \mathbf{KW})^{-1} \mathbf{W} \frac{\partial \mathbf{K}}{\partial \boldsymbol{\theta}_j} \right) \tag{18.33}$$

We can then use the chain rule to get the other parts: $\frac{\partial}{\partial \boldsymbol{\theta}_j} = \frac{\partial}{\partial \hat{f}} \frac{\partial \hat{f}}{\partial \boldsymbol{\theta}_j}$, where:

$$\frac{\partial \hat{f}}{\partial \boldsymbol{\theta}_j} = (\mathbf{I} + \mathbf{WK})^{-1} \frac{\partial \mathbf{K}}{\partial \boldsymbol{\theta}_j} (\mathbf{t} - \sigma(\hat{f}(\mathbf{x})), \tag{18.34}$$

and so we just need to compute:

$$\begin{aligned} &\frac{\partial}{\partial \hat{f}(\mathbf{x}_i)} \log |\mathbf{W} + \mathbf{K}^{-1}| \\ &= \left((\mathbf{I} + \mathbf{WK})^{-1} \mathbf{K} \right)_{ii} \sigma(\hat{f}(\mathbf{x}_i))(1 - \sigma(\hat{f}(\mathbf{x}_i)))(1 - 2\sigma(\hat{f}(\mathbf{x}_i)) \frac{\partial \hat{f}(\mathbf{x}_i)}{\partial \boldsymbol{\theta}_j} \end{aligned} \tag{18.35}$$

Note that this includes the third derivative of the $\sigma(\cdot)$ term!

Putting these three terms together gives us the whole gradient, ready for the conjugate gradient solver.

18.2.3 Implementation

The algorithm can be written out from the previous discussion, but as with the regression case, there are some tricks that can be used to improve the computational time and stability. The main one is that the matrix $(\mathbf{K} + \mathbf{W}^{-1})$ can be inverted using another matrix identity:

$$(\mathbf{K} + \mathbf{W}^{-1})^{-1} = \mathbf{K} - \mathbf{KW}^{\frac{1}{2}} \mathbf{B}^{-1} \mathbf{W}^{\frac{1}{2}} \mathbf{K}, \tag{18.36}$$

where $\cdot^{\frac{1}{2}}$ means the element-wise square root and $\mathbf{B}$ is the symmetric positive definite matrix

$$B = \mathbf{I} + \mathbf{W}^{\frac{1}{2}} \mathbf{KW}^{\frac{1}{2}}. \tag{18.37}$$

To make implementation easier, the algorithm is written out here in these computationally efficient terms:

Gaussian Process Classification

- **To find the maximum by Newton–Raphson iteration:**
  - compute the covariance matrix $\mathbf{K} = k(\mathbf{X}, \mathbf{X}) + \sigma_n \mathbf{I}$ for hyperparameters $\boldsymbol{\theta}$
  - repeat until change < tolerance:
    - $W = -\nabla\nabla \log p(f(\mathbf{x}))$
    - $L = \text{cholesky}(\mathbf{I} + \mathbf{W}^{\frac{1}{2}} \mathbf{KW}^{\frac{1}{2}})$
    - update f using Equation (18.27), with Equation (18.36) giving the form of the inverse matrix
    - change = oldf - f

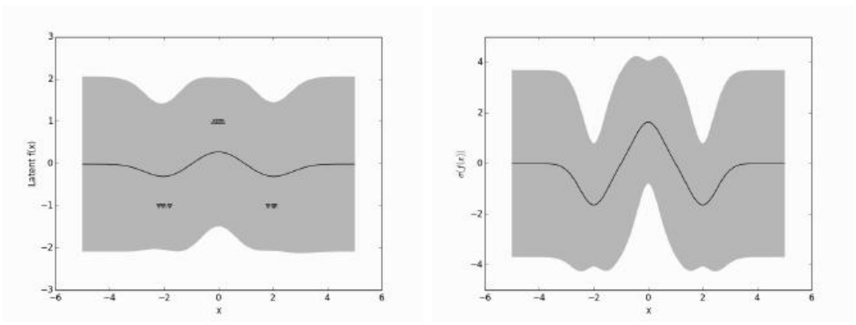

FIGURE 18.8 Gaussian process classification for a very simple dataset (shown plotted in the figure on the left). The latent function can be seen on the left, and the output of the logistic function on the right.

- **To make a prediction:**
  - compute the covariance matrix $\mathbf{k}^* = k(\mathbf{x}^*, \mathbf{X})$
  - compute the covariance matrix $k^{**} = k(\mathbf{x}^*, \mathbf{x}^*)$
  - compute the maximum f^* using the Newton–Raphson iteration algorithm
  - the mean of the process is $\mathbf{k}^* \nabla \log p(f(\mathbf{x}))$
  - solve $\mathbf{Lv} = \mathbf{W}^{\frac{1}{2}} \mathbf{k}^*$ for $\mathbf{v}$
  - the variance is $k^{**} - \mathbf{v}^T \mathbf{v}$
- **To compute the log likelihood and gradient:**
  - compute log likelihood using Equation (18.31)
  - compute $\mathbf{R} = \mathbf{W}^{\frac{1}{2}} \mathbf{B}^{-1} \mathbf{W}^{\frac{1}{2}}$, where $\mathbf{B}$ is defined in Equation (18.37).
  - compute $\mathbf{s}_2 = \frac{\partial}{\partial f(\mathbf{x})} \log q$ using Equation (18.35)
  - for each hyperparameter $\boldsymbol{\theta}_j$:
    - compute gradients of covariance matrix with respect to $\boldsymbol{\theta}_j$
    - compute explicit gradient $s_1 = \frac{\partial}{\partial \boldsymbol{\theta}_j} \log p(\mathbf{t}|\boldsymbol{\theta})$ using Equation (18.33)
    - compute $\mathbf{s}_2 = \frac{\partial \hat{f}}{\partial \boldsymbol{\theta}_j}$ using Equation (18.34)
    - full gradient of log likelihood for $\boldsymbol{\theta}_j$ is $s_1 + \mathbf{s}_2^T \mathbf{s}_3$

Figure 18.8 shows a very simple example of Gaussian process classification. The data consists of a few points at around $x = -2$ and $x = +2$ that belong to one class and a few at around $x = 0$ that belong to the other class.

It is possible to do multi-class classification with GPs. The basic idea is to use a separate latent function for each class (so that the function $f(\mathbf{x})$ gets c times longer for c classes), looking like:

$$(f_1^{C_1}, f_2^{C_1}, \ldots f_n^{C_1}, f_1^{C_2}, f_2^{C_2}, \ldots f_n^{C_2}, \ldots f_1^{C_c}, f_2^{C_c}, \ldots f_n^{C_c}). \tag{18.38}$$

The target vector has to be the same dimension, so it will contain a row of n 1s where the

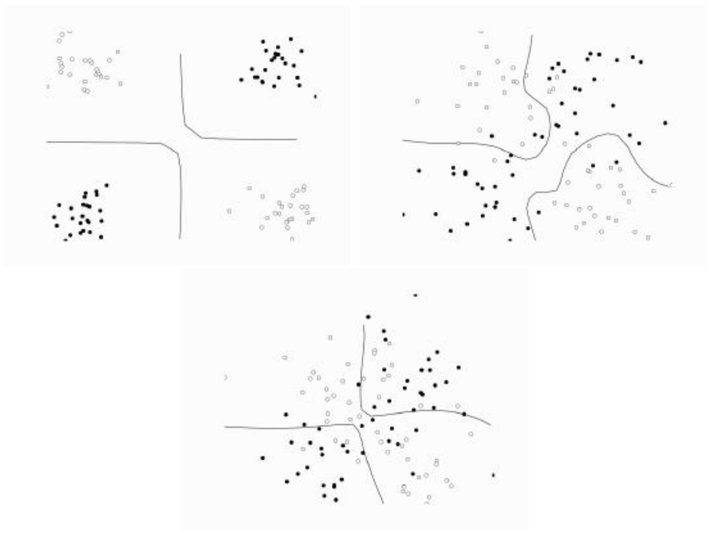

FIGURE 18.9 Gaussian process classification on the modified XOR dataset, with standard deviations $\sigma = 0.1$ (*left*), $\sigma = 0.3$ (middle), $\sigma = 0.4$ (*right*). *The line gives the* $p = 0.5$ *decision boundary.*

f_i for the correct target class are, and 0 everywhere else. The covariance function will then be represented by a set of blocks of the individual covariance matrices. It is also necessary to use soft-max instead of the logistic function to do the 'squashing' of the regression output, which changes the derivatives in the computation of the log likelihood and its gradients. For further details on this, see the references in the Further Reading section.

There has been a lot more work on Gaussian processes over the past 10 years, including far more sophisticated optimisation methods, better ways to perform multi-class classification, and a better understanding of the links between Gaussian processes and neural networks, splines, and many other topics, but they are beyond our scope here: for more information, consult the references in the Further Reading section.

For a fairly simple idea, Gaussian processes do tend to work very well on a wide range of topics, and the way that the covariance function explicitly encodes the correlations that can be seen in the data means that the user has a lot of control. Even in the simple treatment here we have put quite a lot of effort into making the computations numerically stable and relatively fast. However, there is much more that can be done, including methods for approximation to speed things up significantly. Again, the Further Reading section provides more detail.

FURTHER READING

There is a very readable book dedicated solely to Gaussian Processes, which is:

- Carl Edward Rasmussen and Christopher K.I. Williams. *Gaussian Processes for Machine Learning.* MIT Press, Cambridge, MA, USA, 2006.

Another summary that may be useful is:

- D. MacKay. Neural networks and machine learning. *NATO ASI Series, Series F, Computer and Systems Sciences*, 168:133–166, 1998.

and GPs are also covered in:

- D.J.C. MacKay. (Chapter 45) *Information Theory, Inference and Learning Algorithms.* Cambridge University Press, Cambridge, UK, 2003.
- C.M. Bishop. (Section 6.4) *Pattern Recognition and Machine Learning.* Springer, Berlin, Germany, 2006.

PRACTICE QUESTIONS

Problem 18.1 The current implementation only has the squared exponential kernel in. Implement some more of those listed in Section 18.1.5 and experiment with them, particularly with the Palmerston North ozone layer dataset that we saw in Section 4.4.4. You might find the example in Section 5.4.3 of Rasmussen and Williams helpful.

Problem 18.2 Compare the optimisation results with using other optimisers, such as BFGS.

Problem 18.3 A simple version of multiclass classification uses one-against-all classifiers as we did with the SVM. Implement that and see how well it works on the iris dataset.

APPENDIX A

Python

The examples in this book are all written in Python, and the various graphs and results were also created in that language, using the code available via the book website. The purpose of this chapter is to give a brief introduction to using Python, and particularly NumPy, the numerical library for Python.

A.1 INSTALLING PYTHON AND OTHER PACKAGES

The Python language is very compact, but there are huge numbers of extensions and libraries available to make it more suited to a wide variety of tasks. Almost all of the examples in the book use NumPy, a set of numerical libraries, and the figures are produced using Matplotlib. Both of these packages have syntax that is similar to MATLAB®. There are a few places where examples also use SciPy, the scientific programming libraries.

An Internet search will turn up working distributions as self-extracting zip files for the major operating systems, which will include the Python interpreter and all of the packages that are used in the book, amongst others. If you download individual packages, then they generally come with a setup script (`setup.py`) that can be run from a shell. Package webpages generally give instructions.

A.2 GETTING STARTED

There are two ways that Python is commonly used. The first is as an interactive command environment, such as *iPython* or *IDLE*, which are commonly bundled with the Python interpreter. Starting Python with one of these (using Start/IPython in Windows, or by typing `python` at a command prompt in other operating systems) results in a script window with a command prompt (which will be shown as `>>>`). Unlike with C or Java, you can type commands at this prompt and the interpreter will run the commands and display the results, if any, on the screen. You can write functions in a text editor and run them from the command prompt by calling them by name. We'll see more about functions in Section A.3.

As well as iPython there are several other Python IDEs and code editors available for various operating systems. Two nice possibilities are the Java-based IDE *Eclipse* using the extension for Python called *PyDev*, and *Spyder*, which is aimed at exactly the kind of scientific Python that we are doing in this book. Both of these are freely available on the Internet and include all of the usual syntax highlighting and development help. In addition, you can run programs directly, and you can also set up an interactive Python environment so that you can test small pieces of code to see how they work.

The best way to get used to any language is, of course, to write programs in it. There is lots of code in the book and practical programming assignments along the way, but if you haven't used Python before, then it will help if you get used to the language prior to working on the code examples in the book. Section A.3 describes how to get started writing Python programs, but here we will begin by using the command line to see how things work. This can be in iPython or IDLE, by typing `python` at the command prompt, or within the console in the PyDev Eclipse extension or Spyder.

Creating a variable in Python is easy: you give it a name and assign a value. While Python is strongly typed (so that variables that contain integers don't suddenly change to holding strings or floats without being told to) it performs all the declaration and creation of variables for you, unlike lower level languages like C. So typing `>>> a = 3` at the command prompt (note that the `>>>` is the command prompt, so you only actually type `a = 3`) defines `a` as an integer variable and gives it value 3. To see the effect of the integer typing, type `>>> a/2`, and you will see that the answer is `1`. What Python actually does is compute the answer in the most accurate of the types that are included in the calculation, but since `a` is an integer, and so is `2`, it returns the answer as an integer. You can see this using the `type()` function; `type(3/2)` returns `<type 'int'>`. So `>>> a/2.0` will work perfectly well, since the type of `2.0` is a float (`type(3/2.0) = <type 'float'>`. When writing floats, you can abbreviate them to `2.` without the zero if you really want to save typing one character. To see the value of a variable you can just type its name at the command prompt, or use `>>> print a`, or whatever the name of the variable is.

You can perform all of the usual arithmetic operators on numbers, adding them up, etc. Raising numbers to a power is performed by `a**2` or `pow(a,2)`. In fact, you can use Python as a perfectly good calculator at the command line.

Just like in many other languages, comparison is performed using the double equals (==). It returns Boolean values `True` (1) and `False` (0) to tests like `>>> 3 < 4` and `>>> 3 == 4`. The other arithmetic comparisons are also available: `<, <=, >, >=` and these can be chained (so `3<x<6` performs the two tests and only returns `True` if both are true). The not-equal-to test is `!=` or `<>`, and there is another useful comparison: `is` checks if two variables point to the same object. This might not seem important, but Python works by reference, which means that the command `>>> a = b` does not put a copy of the value of `b` into `a`, but rather assigns to `a` a reference to the variable `b`. This can be a trap for the unwary, as will be discussed more shortly. The normal logical operators are slightly unusual in Python, with the normal logical operators using the words `and, or`, and `not`; the symbols `&, |` perform bit-wise and/or. These bit-wise operators are actually quite useful, as we'll see later.

In addition to integer and floating point representations of numbers, Python also deals with strings, which are described by using single or double quotes (' or ") to surround them: `>>> b = 'hello'`. For strings, the `+` operator is overloaded (given a new meaning), which is concatenation: merging the strings. So `>>> 'a' + 'd'` returns the new string `'ad'`.

Having made the basic data types, Python then allows you to combine them into three different basic data structures:

Lists A list is a combination of basic data types, surrounded by square brackets. So `>>> mylist = [0, 3, 2, 'hi']` is a perfectly good list that contains integers and a string. This ability to store different types inside a list gives you a hint that Python handles lists differently to the way other languages handle arrays. This comes about because Python is inherently object-oriented, so that every variable that you make is simply an object, and so a list is just a collection of objects. This is why the type of the object does not matter. It also means that you can have lists of lists without a problem: `>>> newlist = [3, 2, [5, 4, 3], [2, 3, 2]]`.

Accessing particular elements of a list simply requires giving it an index. Like C, but unlike MATLAB®, Python indices start at 0, so `>>> newlist[0]` returns the first element (3). You can also index from the end using a minus sign, so `>>> newlist[-1]` returns the last element, `>>> newlist[-2]` the last-but-one, etc. The length of a list is given by `len`, so `>>> len(newlist)` returns 4. Note that `>>> newlist[3]` returns the list in the 4th location of `newlist` (i.e., `[2, 3, 2]`). To access an element of that list you need an extra index: `>>> newlist[3][1]` returns 3.

A useful feature of Python is the slice operator. This is written as a colon (`:`) and enables you to access sections of a list easily, such as `>>> newlist[2:4]` which returns the elements of `newlist` in positions 2 and 3 (the arguments you use in a slice are inclusive at the start and exclusive at the end, so the second parameter is the first index that is excluded). In fact, the slice can take three operators, which are [start:stop:step], the third element saying what stepsize to use. So `>>> newlist[0:4:2]` returns the elements in locations 0 and 2, and you can use this to reverse the order of a list: `>>> newlist[::-1]`. This last example shows a couple of other refinements of the slice operator: if you don't put a value in for the first number (so it looks like `[:3]`) then the value is taken as 0, and if you don't put a value for the second operator (`[1:]`) then it is taken as running to the end of the list. These can be very useful, especially the second one, since it avoids having to calculate the length of the list every time you want to run through it. `>>> newlist[:]` returns the whole string.

This last use of the slice operator, returning the whole string, might seem useless. However, because Python is object-oriented, all variable names are simply references to objects. This means that copying a variable of type `list` isn't as obvious as it could be. Consider the following command: `>>> alist = mylist`. You might expect that this has made a copy of `mylist`, but it hasn't. To see this, use the following command `>>> alist[3] = 100` and then have a look at the contents of `mylist`. You will see that the 3rd element is now 100. So if you want to copy things you need to be careful. The slice operator lets you make actual copies using: `>>> alist = mylist[:]`. Unfortunately, there is an extra wrinkle in this if you have lists of lists. Remember that lists work as references to objects. We've just used the slice operator to return the values of the objects, but this only works for one level. In location 2 of `newlist` is another list, and the slice operator just copied the reference to that embedded list. To see this, perform `>>> blist = newlist[:]` and then `>>> blist[2][2] = 100` and have a look at `newlist` again. What we've done is called a shallow copy, to copy everything (known as a deep copy) requires a bit more effort. There is a `deepcopy` command, but to get to it we need to `import` the `copy` module using `>>> import copy` (we will see more about importing in Section A.3.1). Now we can call `>>> clist = copy.deepcopy(newlist)` and we finally have a copy of a complete list.

There are a variety of functions that can be applied to lists, but there is another interesting feature of the fact that they are objects. The functions (methods) that can be used are part of the object class, so they modify the list itself and do not return a new list (this is known as working in place). To see this, make a new list `>>> list = [3, 2, 4, 1]` and suppose that you want to print out a list of the numbers sorted into order. There is a function `sort()` for this, but the obvious `>>> print list.sort()` produces the output `None`, meaning that no value was returned. However, the two commands `>>> list.sort()` followed by `>>> print list` do exactly what is required. So functions on lists modify the list, and any future operations will be applied to this modified list.

Some other functions that are available to operate on lists are:

`append(x)` adds `x` to the end of the list

`count(x)` counts how many times `x` appears in the list

`extend(L)` adds the elements in list `L` to the end of the original list

`index(x)` returns the index of the first element of the list to match `x`

`insert(i, x)` inserts element `x` at location `i` in the list, moving everything else along

`pop(i)` removes the item at index `i`

`remove(x)` deletes the first element that matches `x`

`reverse()` reverses the order of the list

`sort()` we've already seen

You can compare lists using `>>> a==b`, which works elementwise through the list, comparing each element against the matching one in the second list, returning True if the test is true for each pair (and the two lists are the same length), and False otherwise.

Tuples A tuple is an immutable list, meaning that it is read-only and doesn't change. Tuples are defined using round brackets, e.g., `>>> mytuple = (0, 3, 2, 'h')`. It might seem odd to have them in the language, but they are useful if you want to create lists that cannot be modified, especially by mistake.

Dictionaries In the list that we saw above we indexed elements by their position within the list. In a dictionary you assign a key to each entry that you can use to access it. So suppose you want to make a list of the number of days in each month. You could use a dictionary (shown by the curly braces): `>>> months = {'Jan': 31, 'Feb': 28, 'Mar': 31}` and then you access elements of the dictionary using their key, so `>>> months['Jan']` returns 31. Giving an incorrect key results in an exception error.

The function `months.keys()` returns a list of all the keys in the dictionary, which is useful for looping over all elements in a dictionary. The `months.values()` function returns a list of values instead, while `months.items()` gives a list of tuples containing everything. There are lots of other things you can do with dictionaries, and we shall see some of them when we use the dictionary in Chapter 12.

There is one more data type that is built directly into Python, and that is the `file`. This makes reading from and writing to files very simple in Python: files are opened using `>>> input = open('filename')`, closed using `>>> input.close()` and reading and writing are performed using `readlines()` (and `read()`, and `writelines()` and `write()`). There are also `readline()` and `writeline()` functions, that read and write one line at a time.

A.2.1 Python for MATLAB® and R users

With the NumPy package that we are using there are a great many similarities between MATLAB® or R and Python. There are useful comparison websites for both MATLAB® and R, but the main thing that you need to be aware of is that indexing starts at 0 instead of 1 and elements of arrays are accessed with square brackets instead of round ones. After that, while there are differences, the similarity between the three languages is striking.

A.3 CODE BASICS

Python has a fairly small set of commands and is designed to be fairly small and simple to use. In this section we'll go over the basic commands and other programming details. There are lots of good resources available for getting started with Python; a few books are listed at the end of the chapter, and an Internet search will provide plenty of other resources.

A.3.1 Writing and Importing Code

Python is a scripting language, meaning that everything can be run interactively from the command line. However, when writing any reasonable sized piece of code it is better to write it in a text editor or IDE and then run it. The programming GUIs provide their own code writing editors, but you can also use any text editor available on your machine. It is a good idea to use one that is consistent in its tabbing, since the white space indentation is how Python blocks code together.

The file can contain a script, which is simply a series of commands, or a set of functions and classes. In either case it should be saved with a `.py` extension, which Python will compile into a `.pyc` file when you first load it. Any set of commands or functions is known as a module in Python, and to load it you use the `import` command. The most basic form of the command is `import name`. If you import a script file then Python will run it immediately, but if it is a set of functions then it will not run anything.

To run a function you use `>>> name.functionname()`, where `name` is the name of the module and `functionname` the relevant function. Arguments can be passed as required in the brackets, but even if no arguments are passed, then the brackets are still needed. Some names get quite long, so it can be useful to use `import x as y`, which means that you can then use `>>> y.functionname()` instead.

When developing code at a command line there is one slightly irritating feature of Python, which is that `import` only works once for a module. Once a module has been imported, if you change the code and want Python to work on the new version, then you need to use `>>> reload(name)`. Using `import` will not give any error messages, but it will not work, either.

Many modules contain several subsets, so when importing you may need to be more specific. You can import particular parts of a module in this way using `from x import y`, or to import everything use `from x import *`, although this is rarely a good idea as some of the modules are very large. Finally, you can specify the name that you want to import the module as, by using `from x import y as z`.

Program code also needs to import any modules that it uses, and these are usually declared at the top of the file (although they don't need to be, but can be added anywhere). There is one other thing that might be confusing, which is that Python uses the `pythonpath` variable to tell it where to look for code. Eclipse doesn't include other packages in your current project on the path, and so if you want it to find those packages, you have to add them to the path using the Properties menu item while Spyder has it in the 'Spyder' menu. If you are not using either or these, then you will need to add modules to the path. This can be done using something like:

```
import sys
sys.path.append('mypath')
```

A.3.2 Control Flow

The most obviously strange thing about Python for those who are used to other programming languages is that the indentation means something: white space is the way that blocks of code are shown. So if you have a loop or other construct, then the equivalent of `begin ... end` or the braces { } in other languages is a colon (:) after the keyword and indented commands following on. This looks quite strange at first, but is actually quite nice once you get used to it. The other thing that is unusual is that you can have an (optional) `else` clause on loops. This clause runs when the loop terminates normally. If you break out of a loop using the `break` command, then the `else` clause does not run.

The control structures that are available are `if`, `for`, and `while`. The `if` statement syntax is:

```
if statement:
    commands
elif:
    commands
else:
    commands
```

The most common loop is the `for` loop, which differs slightly from other languages in that it iterates over a list of values:

```
for var in set:
    commands
else:
    commands
```

There is one very useful command that goes with this `for` loop, which is the `range` command, which produces a list output. Its most basic form is simply `>>> range(4)`, which produces the list `[0, 1, 2, 3]`. However, it can also take 2 or 3 arguments, and works in the same way as in the slice command, but with commas between them instead of colons: `>>> range(start,stop,step)`. This can include going down instead of up a list, so `>>> range(5,-3,-2)` produces `[5, 3, 1, -1]` as output.

Finally, there is a `while` loop:

```
while condition:
    commands
else:
    commands
```

A.3.3 Functions

Functions are defined by:

```
def name(args):
    commands
    return value
```

The `return value` line is optional, but enables you to return values from the function (otherwise it returns `None`). You can list several things to return in the line with commas between them, and they will all be returned. Once you have defined a function you can call it from the command line and from within other functions. Python is case sensitive, so with both function names and variable names, `Name` is different to `name`.

As an example, here is a function that computes the hypotenuse of a triangle given the other two distances (`x` and `y`). Note the use of '#' to denote a comment:

```
def pythagoras(x,y):
    """ Computes the hypotenuse of two arguments"""
    h = pow(x**2+y**2,0.5)
    # pow(x,0.5) is the square root
    return h
```

Now calling `pythagoras(3,4)` gets the expected answer of `5.0`. You can also call the function with the parameters in any order provided that you specify which is which, so `pythagoras(y=4,x=3)` is perfectly valid. When you make functions you can allow for default values, which means that if fewer arguments are presented the default values are given. To do this, modify the function definition line: `def pythagoras(x=3,y=4):`

A.3.4 The doc String

The help facilities within Python are accessed by using `help()`. For help on a particular module, use `help('modulename')`. (So using `help(pythagorus)` in the previous example would return the description of the function that is given there). A useful resource for most code is the `doc` string, which is the first thing defined within the function, and is a text string enclosed in three sets of double quotes (`"""`). It is intended to act as the documentation for the function or class. It can be accessed using `>>> print functionname.__doc__`. The Python documentation generator `pydoc` uses these strings to automatically generate documentation for functions, in the same way that `javadoc` does.

A.3.5 `map` and `lambda`

Python has a special way of performing repeated function calls. If you want to apply the same function to every element of a list you don't need to loop over the elements of the list, but can instead use the `map` command, which looks like `map(function,list)`. This applies the function to every element of the list. There is one extra tweak, which is the fact that the function can be anonymous (created just for this job without needing a name) by using the `lambda` command, which looks like `lambda args : command`. A `lambda` function can only execute one command, but it enables you to write very short code to do relatively complicated things. As an example, the following instruction takes a list and cubes each number in it and adds 7:

```
map(lambda x:pow(x,3)+7,list)
```

Another way that `lambda` can be used is in conjunction with the `filter` command. This returns elements of a list that evaluate to `True`, so:

```
filter(lambda x:x>=2,list)
```

returns those elements of the list that are greater than or equal to 2. NumPy provides simpler ways to do these things for arrays of numbers, as we shall see.

A.3.6 Exceptions

Like other modern languages, Python allows for the trapping of exceptions. This is done through the `try ... except ... else` and `try... finally` constructions. This example shows the use of the most common version. For more details, including the types of exceptions that are defined, see a Python programming book.

```
try:
   x/y
except ZeroDivisonError:
   print "Divisor must not be 0"
except TypeError:
   print "They must be numbers"
except:
   print "Something unspecified went wrong"
else:
   print "Everything worked"
```

A.3.7 Classes

For those that wish to use it in this way, Python is fully object-oriented, and classes are defined (with their constructor) by:

```
class myclass(superclass):

   def ___init___(self,args):

   def functionname(self,args):
```

If a superclass is not specified, then the class does not inherit from elsewhere. The `__init__(self,args)` function is the constructor for the class. There can also be a destructor `__del__(self)`, although they are rarely used. Accessing methods from the class uses the `classname.functionname()` syntax. The `self` argument can be ignored in all function calls, since Python fills it in for you, but it does need to be specified in the function definition. Many of the examples in the book are based on classes provided on the book website. You need to be aware that you have to create an instance of the class before you can run it. There is one extra thing that can catch the unwary. If you have imported a module within a program and then you change the code of the module that you have imported, reloading the program won't reload the module. So to import and run the changed module you need to use:

```
import myclass
var = myclass.myclass()
var.function()
```

and if there is a module within there that you expect to change (for example, during testing or further development, you modify it a little to include:

```
import myclass
reload(myclass)
var = myclass.myclass()
var.function()
```

A.4 USING NUMPY AND MATPLOTLIB

Most of the commands that are used in this book actually come from the NumPy and Matplotlib packages, rather than the basic Python language. More specialised commands are described thoughout the book in the places where they become relevant. There are lots of examples of performing tasks using the various functions within NumPy on its website. Getting information about functions within NumPy is generally done using `help(np.functionname)` such as `help(np.dot)`.

NumPy has a base collection of functions and then additional packages that have to be imported as well if you want to use them. To import the NumPy base library and get started you use:

```
>>> import numpy as np
```

A.4.1 Arrays

The basic data structure that is used for numerical work, and by far the most important one for the programming in this book, is the `array`. This is exactly like multi-dimensional arrays (or matrices) in any other language; it consists of one or more dimensions of numbers or chars. Unlike Python lists, the elements of the array all have the same type, which can be Boolean, integer, real, or complex numbers.

Arrays are made using a function call, and the values are passed in as a list, or set of lists for higher dimensions. Here are one-dimensional and two-dimensional arrays (which are effectively arrays of arrays) being made. Arrays can have as many dimensions as you like up to a language limit of 40 dimensions, which is more than enough for this book.

```
>>> myarray = np.array([4,3,2])
>>> mybigarray = np.array([[3, 2, 4], [3, 3, 2], [4, 5, 2]])
>>> print myarray
[4 3 2]
>>> print mybigarray
[[3 2 4]
 [3 3 2]
 [4 5 2]]
```

Making arrays like this is fine for small arrays where the numbers aren't regular, but there are several cases where this is not true. There are nice ways to make a set of the more interesting arrays, such as those shown next.

Array Creation Functions

`np.arange()` Produces an array containing the specified values, acting as an array version of `range()`. For example, `np.arange(5) = array([0, 1, 2, 3, 4])` and `np.arange(3,7,2) = array([3, 5])`.

`np.ones()` Produces an array containing all ones. For both `np.ones()` and `np.zeros()` you need two sets of brackets when making arrays of more than one dimension. `np.ones(3) = array([ 1., 1., 1.])` and `np.ones((3,4)) =`

```
array([[ 1.,  1.,  1., 1,]
[ 1.,  1.,  1., 1.]
[ 1.,  1.,  1., 1.]])
```

You can specify the type of arrays using `a = np.ones((3,4),dtype=float)`. This can be useful to ensure that you don't run into problems with integer casting, although NumPy is fairly good at casting things as floats.

`np.zeros()` Similar to `np.ones()`, except that all elements of the matrix are zero.

`np.eye()` Produces the identity matrix, i.e., the 2D matrix that is zero everywhere except down the leading diagonal, where it is one. Given one argument it produces the square identity: `np.eye(3) =`

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

while with two arguments it fills spare rows or columns with zeros: `np.eye(3,4) =`

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

`np.linspace(start,stop,npoints)` Produces a matrix with linearly spaced elements. The nice thing is that you specify the number of elements, not the spacing. `np.linspace(3,7,3) = array([ 3., 5., 7.])`

`np.r_[]` and `np.c_[]` Perform row and column concatenation, including the use of the slice operator: `np.r_[1:4,0,4] = array([1, 2, 3, 0, 4])`. There is also a variation on `np.linspace()` using a `j` in the last entry: `np.r_[2,1:7:3j] = array([ 2. , 1. , 4. , 7.])`. This is another nice feature of NumPy that can be used with `np.arange()` and `np.meshgrid()` as well. The `j` on the last value specifies that you want 3 equally spaced points starting at 0 and running up to (and including) 7, and the function works out the locations of these points for you. The column version is similar.

The array `a` used in the next set of examples was made using `>>> a = np.arange(6).reshape(3,2)`, which produces:

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

Indexing elements of an array is performed using square brackets '[' and ']', remembering that indices start from 0. So `a[2,1]` returns `5` and `a[:,1]` returns `array([1, 3, 5])`. We can also get various pieces of information about an array and change it in a variety of different ways, as follows.

Getting information about arrays, changing their shape, copying them

---

`np.ndim(a)` Returns the number of dimensions (here `2`).

`np.size(a)` Returns the number of elements (here `6`).

`np.shape(a)` Returns the size of the array in each dimension (here `(3, 2)`). You can access the first element of the result using `shape(a)[0]`.

`np.reshape(a,(2,3))` Reshapes the array as specified. Note that the new dimensions are in brackets. One nice thing about `np.reshape()` is that you can use '-1' for 1 dimension within the reshape command to mean 'as many as is required'. This saves you doing the multiplication yourself. For this example, you could use `np.reshape(a,(2,-1))` or `np.reshape(a,(-1,2))`.

`np.ravel(a)` Makes the array one-dimensional (here `array([0, 1, 2, 3, 4, 5])`).

`np.transpose(a)` Compute the matrix transpose. For the example:

```
[[0 2 4]
 [1 3 5]]
```

`a[::-1]` Reverse the elements of each dimension.

`np.min(), np.max(a), np.sum(a)` Returns the smallest or largest element of the matrix, or the sum of the elements. Often used to sum the rows or columns using the `axis` option: `np.sum(axis=0)` for columns and `np.sum(axis=1)` for rows.

`np.copy()` Makes a deep copy of a matrix.

Many of these functions have an alternative form that like `a.min()` which returns the minimum of array `a`. This can be useful when you are dealing with single matrices. In particular, the shorter version of the transpose operator, `a.T`, can save a lot of typing.

Just like the rest of Python, NumPy generally deals with references to objects, rather than the objects themselves. So to make a copy of an array you need to use `c = a.copy()`.

Once you have defined matrices, you need to be able to add and multiply them in different ways. As well as the array `a` used above, for the following set of examples two other arrays `b` and `c` are needed. They have to have sizes relating to array `a`. Array `b` is the same size as `a` and is made by `>>> b = np.arange(3,9).reshape(3,2)`, while `c` needs to have the same inner dimension; that is, if the size of `a` is `(x, 2)` then the size of `c` needs to be `(2, y)` where the values of `x` and `y` don't matter. For the examples `>>> c = np.transpose(b)`. Here are some of the operations you can perform on arrays and matrices:

Operations on arrays

`a+b` Matrix addition. Output for the example is:

```
array([[ 3,  5],
       [ 7,  9],
       [11, 13]])
```

`a*b` Element-wise multiplication. Output:

```
array([[ 0,  4],
       [10, 18],
       [28, 40]])
```

`np.dot(a,c)` Matrix multiplication. Output:

```
array([[ 4,  6,  8],
       [18, 28, 38],
       [32, 50, 68]])
```

`pow(a,2)` Compute exponentials of elements of matrix (a Python function, not a NumPy one). Output:

```
array([[ 0,  1],
       [ 4,  9],
       [16, 25]])
```

`pow(2,a)` Compute number raised to matrix elements (a Python function, not a NumPy one). Output:

```
array([[ 1,  2],
       [ 4,  8],
       [16, 32]])
```

Matrix subtraction and element-wise division are also defined, but the same trap that we saw earlier can occur with division, namely that `a/3` returns an integer not a float if `a` is an array of integers.

There is one more very useful command on arrays, which is the `np.where()` command. This has two forms: `x = np.where(a>2)` returns the indices where the logical expression is true in the variable `x`, while `x = np.where(a>2,0,1)` returns a matrix the same size as `a` that contains 0 in those places where the expression was true in `a` and 1 everywhere else. To chain these conditions together you have to use the bitwise logical operations, so that `indices = np.where((a[:,0]>3) | (a[:,1]<3))` returns a list of the indices where either of these statements is true.

A.4.2 Random Numbers

There are some good random number features within NumPy, which you access in `np.random` after importing NumPy. To find out about the functions use `help(np.random)` once NumPy has been imported, but the more useful functions are:

`np.random.rand(matsize)` produces uniformly distributed random numbers between 0 and 1 in an array of size `matsize`

`np.random.randn(matsize)` produces zero mean, unit variance Gaussian random numbers

`np.random.normal(mean,stdev,matsize)` produces Gaussian random numbers with specifed mean and standard deviation

`np.random.uniform(low,high,matsize)` produces uniform random numbers between low and high

`np.random.randint(low,high,matsize)` produces random integer values between low and high

A.4.3 Linear Algebra

NumPy has a reasonable linear algebra package that performs standard linear algebra functions. The functions are available as `np.linalg.inv(a)`, etc., where `a` is an array and possible functions are (if you don't know what they all are, don't worry: they will be defined where they are used in the book):

`np.linalg.inv(a)` Compute the inverse of (square) array `a`

`np.linalg.pinv(a)` Compute the pseudo-inverse, which is defined even if `a` is not square

`np.linalg.det(a)` Compute the determinant of `a`

`np.linalg.eig(a)` Compute the eigenvalues and eigenvectors of `a`

A.4.4 Plotting

The plotting functions that we will be using are in the Matplotlib package (also known as pylab, and which we will import as `import pylab as pl`). These are designed to look exactly like the MATLAB® plotting functions. The entire set of functions, with examples, are given on the Matplotlib webpage, but the two most important ones that we will need are `pl.plot` and `pl.hist`. When producing plots they sometimes do not appear. This is usually because you need to specify the command `>>> pl.ion()` which turns interactive plotting on. If you are using Matplotlib within Eclipse it has a nasty habit of closing all of the display windows when the program finishes. To get around this, issue a `show()` command at the end of your function.

The basic plotting commands of Matplotlib are demonstrated here, for more advanced plotting facilities see the package webpage.

The following code (best typed into a file and executed as a script) computes a Gaussian function for values -2 to 2.5 in steps of 0.01 and plots it, then labels the axes and gives the figure a title. The output of running it is shown in Figure A.1.

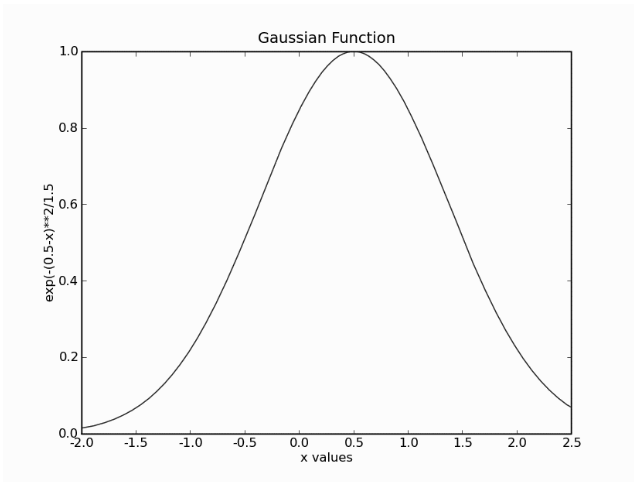

FIGURE A.1 The Matplotlib package produces useful graphical output, such as this plot of the Gaussian function.

```
import pylab as pl
import numpy as np

gaussian = lambda x: exp(-(0.5-x)**2/1.5)
x = np.arange(-2,2.5,0.01)
y = gaussian(x)
pl.ion()
pl.figure()
pl.plot(x,y)
pl.xlabel('x values')
pl.ylabel('exp(-(0.5-x)**2/1.5')
pl.title('Gaussian Function')
pl.show()
```

There is another very useful way to make arrays in NumPy, which is `np.meshgrid()`. It can be used to make a set of indices for a grid, so that you can quickly and easily access all the points within the grid. This has many uses for us, not least of which is to find a classifier line, which can be done using `np.meshgrid()` and then drawn using `pl.contour()`:

```
pl.figure()
step=0.1
f0,f1  = np.meshgrid(np.arange(-2,2,step), np.arange(-2,2,step))

# Run a classifier algorithm
out = classifier(np.c_[np.ravel(f0), np.ravel(f1)],soft=True).T
out = out.reshape(f0.shape)
```

```
pl.contourf(f0, f1, out)
```

A.4.5 One Thing to Be Aware of

NumPy is mostly great to use, and extremely powerful. However, there is one thing that I still find annoying on occasion, and that is the two different types of vector. The following set of commands typed at the command line and the output produced show the problem:

```
>>> a = np.ones((3,3))
>>> a
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> np.shape(a)
(3, 3)
>>> b = a[:,1]
>>> b
array([ 1.,  1.,  1.])
>>> np.shape(b)
(3,)
>>> c = a[1,:]
>>> np.shape(c)
(3,)
>>> print c.T
>>> c
array([ 1.,  1.,  1.])
>>> c.T
array([ 1.,  1.,  1.])
```

When we use a slice operator and only index a single row or column, NumPy seems to turn it into a list, so that it stops being either a row or a column. This means that the transpose operator doesn't do anything to it, and also means that some of the other behaviour can be a little odd. It's a real trap for the unwary, and can make for some interesting bugs that are hard to find in programs. There are a few ways around the problem, of which the two simplest are shown below: either listing a start and end for the slice even for a single row or column, or explicitly reshaping it afterwards.

```
>>> c = a[0:1,:]
>>> np.shape(c)
(1, 3)
>>> c = a[0,:].reshape(1,len(a))
>>> np.shape(c)
(1, 3)
```

FURTHER READING

Python has become incredibly popular for both general computing and scientific computing. Because writing extension packages for Python is simple (it does not require any special programming commands: any Python module can be imported as a package, as can packages written in C), many people have done so, and made their code available on the Internet. Any search engine will find many of these, but a good place to start is the Python Cookbook website.

If you are looking for more complete introductions to Python, some of the following may be useful:

- M.L. Hetland. *Beginning Python: From Novice to Professional*, 2nd edition, Apress Inc., Berkeley, CA, USA, 2008.
- G. van Rossum and F.L. Drake Jr., editors. *An Introduction to Python.* Network Theory Ltd, Bristol, UK, 2006.
- W.J. Chun. *Core Python Programming.* Prentice-Hall, New Jersey, USA, 2006.
- B. Eckel. *Thinking in Python.* Mindview, La Mesa, CA, USA, 2001.
- T. Oliphant. Guide to NumPy, e-book, 2006. The official guide to NumPy by its creator.

PRACTICE QUESTIONS

Problem A.1 Make an array `a` of size 6×4 where every element is a 2.

Problem A.2 Make an array `b` of size 6×4 that has 3 on the leading diagonal and 1 everywhere else. (You can do this without loops.)

Problem A.3 Can you multiply these two matrices together? Why does `a * b` work, but not `dot(a,b)`?

Problem A.4 Compute `dot(a.transpose(),b)` and `dot(a,b.transpose())`. Why are the results different shapes?

Problem A.5 Write a function that prints some output on the screen and make sure you can run it in the programming environment that you are using.

Problem A.6 Now write one that makes some random arrays and prints out their sums, the mean value, etc.

Problem A.7 Write a function that consists of a set of loops that run through an array and count the number of ones in it. Do the same thing using the `where()` function (use `info(where)` to find out how to use it).

Chapman & Hall/CRC
Machine Learning & Pattern Recognition Series

Machine Learning: An Algorithmic Perspective, Second Edition helps you understand the algorithms of machine learning. It puts you on a path toward mastering the relevant mathematics and statistics as well as the necessary programming and experimentation.

New to the Second Edition

- Two new chapters on deep belief networks and Gaussian processes
- Reorganization of the chapters to make a more natural flow of content
- Revision of the support vector machine material, including a simple implementation for experiments
- New material on random forests, the perceptron convergence theorem, accuracy methods, and conjugate gradient optimization for the multi-layer perceptron
- Additional discussions of the Kalman and particle filters
- Improved code, including better use of naming conventions in Python

The text strongly encourages you to practice with the code. Each chapter includes detailed examples along with further reading and problems. All of the Python code used to create the examples is available on the author's website.

Features

- Reflects recent developments in machine learning, including the rise of deep belief networks
- Presents the necessary preliminaries, including basic probability and statistics
- Discusses supervised learning using neural networks
- Covers dimensionality reduction, the EM algorithm, nearest neighbor methods, optimal decision boundaries, kernel methods, and optimization
- Describes evolutionary learning, reinforcement learning, tree-based learners, and methods to combine the predictions of many learners
- Examines the importance of unsupervised learning, with a focus on the self-organizing feature map
- Explores modern, statistically based approaches to machine learning

CRC Press
Taylor & Francis Group
an informa business
www.crcpress.com

6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
711 Third Avenue
New York, NY 10017
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK

K18981
ISBN: 978-1-4665-8328-3
90000
9 781466 583283
www.crcpress.com